LE MODÈLE ROOM : FORMALISATION DE SA SÉMANTIQUE STATIQUE ET SON APPLICATION AU DÉVELOPPEMENT D'UN LOGICIEL DE POURSUITE DE SATELLITES

par

Marc Normandeau

mémoire présenté au Département de mathématiques
et d'informatique en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, mars 1997

Sommaire

Le domaine des systèmes contrôle-commande en temps réel requiert beaucoup de travail au niveau de la spécification du comportement. La demande croissante de ce type de logiciel et les délais de production de plus en plus courts forcent l'utilisation d'outils de développement. Ces derniers doivent permettre d'accélérer le processus de développement et d'en améliorer la qualité. Le modèle ROOM offre un cadre de travail pour la conception et la vérification de systèmes temps-réel.

L'outil ObjecTime implante cette technique de modélisation. Il permet de spécifier graphiquement un système, de le simuler et d'en générer le code pour une plate-forme cible. Ce formalisme graphique est accompagné d'un langage textuel (ROOM). Cependant, à cause de la nature du domaine des systèmes contrôle-commande en temps réel, une vérification formelle de la sémantique statique est nécessaire. Elle permet d'accroître la fiabilité de l'outil en éliminant les erreurs dues au langage dans les modèles générés. Sous un autre aspect, l'arrivée récente de cette méthodologie sur le marché implique un manque de maturité. Il lui est donc important d'acquérir de l'expérience sur des problèmes réels.

Les résultats de ce mémoire sont un logiciel de poursuite de satellites développé selon ROOM et une spécification formelle de la sémantique statique du langage ROOM. Une étude de cas a été faite pour vérifier l'application du modèle ROOM à un exemple concret. Le problème choisi n'est pas qu'un simple exemple académique, car le logiciel résultant a été installé dans une station de poursuite de satellites pour vérifier sa qualité. La

formalisation du langage est faite à l'aide des grammaires attribuées. Les grammaires attribuées permettent de définir formellement les propriétés des langages contextuels ou non contextuels. Le mémoire présente également une étude préliminaire des outils d'aide à l'implantation d'un compilateur pour le langage et fournit des recommandations quant à ceux-ci.

Remerciements

D'abord et avant tout, je tiens à remercier de façon particulière mon directeur de recherche, le professeur Michel Barbeau, qui m'a guidé tout au long de ma recherche. Ses précieux conseils et son opinion m'ont été d'un grand secours. Grâce à lui, j'ai pu profiter d'un environnement de recherche idéal. Son soutien continu a rendu ma tâche à la fois plus facile et plus agréable. Je lui serai donc, à juste titre, toujours reconnaissant. Merci également à tous les professeurs et les étudiants du Département de mathématiques et d'informatique de l'Université de Sherbrooke. Plus spécialement, j'aimerais remercier le travail de M. Jacques Haguel, M. Jean Goulet et M. Marc Frappier pour l'évaluation de mon mémoire. Leurs observations et commentaires me furent très précieux.

Les auteurs de l'étude de cas tiennent à exprimer leur reconnaissance à M. Francis Bordeleau de l'Université Carleton pour les discussions fort utiles qu'ils ont eues au sujet du modèle ROOM du logiciel de poursuite de satellites.

Cette maîtrise a été réalisée dans le cadre d'un projet conjoint avec le Centre de recherche informatique de Montréal et subventionnée par le Conseil de recherches en sciences naturelles et en génie du Canada.

Table des matières

Liste des tableaux

Liste des figures

Introduction

Le développement de systèmes temps-réels connaît actuellement une croissance importante. La miniaturisation du matériel, ainsi que la réduction de son coût comptent parmies les principales causes. L'accroissement de la production de logiciels dédiés à cette fin n'implique toutefois pas une hausse de leur fiabilité. L'utilisation de méthodologies et d'outils de développement sont des bons moyens d'augmenter la qualité des systèmes produits. Cependant, le domaine des applications temps-réel comporte des particularités qui nécessitent des méthodologies et des outils spéciaux. C'est pourquoi des gens se penchent sur des méthodes pour améliorer les moyens de production de telles applications.

La méthodologie Real-Time Object-Oriented Modeling (ROOM) [26] a été créée dans le but précis d'améliorer et d'accélérer la production de logiciels dédiés au domaine des systèmes temps-réel. Par contre, comme cette méthodologie est relativement récente et qu'elle s'attaque à des problèmes souvent qualifiés de critiques, il est primordial d'assurer son efficacité. Pour ce faire, il est possible de procéder de deux façons, soit par une vérification théorique, soit par une vérification pratique.

Le présent mémoire aborde le problème de vérification de la méthodologie ROOM selon les deux approches susmentionnées. Dans un premier temps, un logiciel de poursuite de satellites en temps réel, appelé SatSy, a été conçu selon la méthodologie ROOM. Pour résoudre le problème de la localisation des satellites dans l'espace, le modèle mathématique à deux corps est utilisé [5]. L'analyse et la conception du logiciel sont faites selon la méthodologie ROOM et à l'aide de l'outil ObjecTime [18]. L'implantation est faite en

C++ sur la plate-forme QNX [21]. Dans un deuxième temps, la vérification formelle du langage ROOM est faite à l'aide des grammaires attribuées selon la référence [31].

Les résultats de ce mémoire se traduisent par un logiciel orienté objet de poursuite de satellites en temps réel, une grammaire attribuée du langage ROOM ainsi qu'une étude préliminaire à l'implantation de cette dernière. La première réalisation est d'autant plus intéressante qu'elle est effectivement utilisée dans une station de communications par satellites. La formalisation de la sémantique statique par la grammaire a permis de relever des omissions et des incohérences dans le langage. L'étude préliminaire a permis de recommander des outils propres à l'implantation de la grammaire atttribuée.

La suite de ce mémoire comporte quatre chapitres et une conclusion. Le chapitre 1 offre une introduction au modèle ROOM ainsi qu'un aperçu d'autres méthodologies dédiées au domaine des systèmes temps-réel. Le chapitre 2 détaille tout ce qui entoure la réalisation de SatSy soit la problématique, l'analyse, la conception et l'implantation. Pour le chapitre 3, le résultat de la formalisation de la sémantique statique est présenté sous forme d'une grammaire attribuée. Le quatrième chapitre fait état d'une étude préliminaire à l'implantation de la grammaire attribuée obtenue.

Chapitre 1

Le modèle ROOM

Ce chapitre introduit le modèle ROOM. La première section aborde ses principales caractéristiques. Elles sont présentées sous trois angles: la structure, les communications et le comportement. La deuxième section compare le modèle ROOM à d'autres méthodologies et langages dédiés aux applications temps-réel dont OORT [20] et UMLRT [28], [29].

1.1 Les particularités du modèle ROOM

Le modèle ROOM est une technique de modélisation. Un système est spécifié à l'aide d'une représentation graphique simple. La notation est minimale et bien adaptée aux systèmes temps-réel. L'outil ObjecTime [18] est le logiciel qui permet la création, la modification, la simulation et l'implantation d'un modèle. Une conception peut être observée selon deux points de vue différents: la structure et le comportement. La structure présente l'architecture du modèle en définissant les éléments qui le composent et les liens qui les unissent. Le comportement montre comment le système peut évoluer dans le temps. Il est affecté par le temps et par certains événements. Les événements sont provoqués par le système ou par son environnement. Toutes les communications se font à l'aide de messages qui sont échangés de façon synchrone ou asynchrone.

Dès qu'un modèle cohérent est spécifié, il peut être exécuté dans un environnement de simulation. Une des particularités intéressantes de l'outil de simulation est le fait qu'un modèle incomplet est exécutable. Ainsi, des parties du modèle peuvent être testées alors qu'elles ne sont pas encore complètement spécifiées. L'outil permet également la création automatique du code d'implantation pour une plate-forme cible. La prochaine sous-section présente le modèle plus en détail en terme de structure, de communications et de comportement.

1.1.1 La modélisation structurelle

Le modèle ROOM est basé sur trois catégories d'entités structurelles: les acteurs, les protocoles et les objets de données. Un modèle complet est constitué d'un ensemble de telles entités bien que, sous sa forme la plus simple, il puisse contenir qu'un seul acteur. Un acteur est un objet actif dont le comportement est significatif. Un protocole représente un ensemble de messages qui sont transmis d'un acteur à l'autre. Enfin, un objet de données est la représentation de base de l'information. Chacune de ces entités est assujettie au paradigme orienté objet. Les définitions de types sont des classes et les incarnations de ces classes sont des acteurs, des ports ou des objets de données. L'accès à ces objets n'est possible que par référence. L'héritage, qui est un concept clé du paradigme orienté objet, est applicable aux trois catégories d'entités. ROOM adopte un système d'héritage non-strict qui permet la création, la modification et la destruction des propriétés d'une classe. L'héritage multiple n'est cependant pas permis. La structure du modèle sera maintenant présentée plus en détail en abordant les classes d'acteurs, les références d'acteurs et les objets de données.

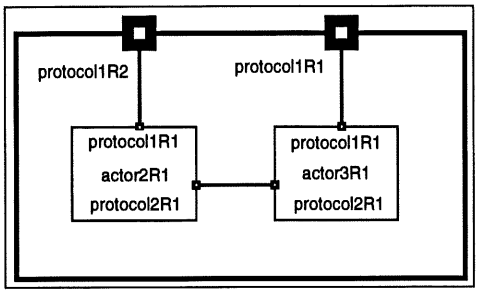

FIG. 1 – *Un diagramme de structure ROOM*

Les classes d'acteurs

Le concept d'acteur est au coeur du modèle ROOM. Les acteurs constituent les objets actifs d'une application. Les capteurs, les actionneurs et les contrôleurs d'un système temps-réel sont modélisés par des acteurs. Ce sont des éléments indépendants d'un processus qui peuvent tous être réutilisés dans plusieurs contextes. C'est pourquoi les acteurs sont encapsulés et les services qu'ils offrent sont disponibles uniquement par leur interface. De cette façon, ils ignorent la présence des acteurs qui les entourent ce qui entraîne un faible couplage. L'interface d'une classe d'acteurs est l'ensemble des ports qui se situent à sa limite extérieure. Elle permet la communication avec des acteurs externes. La structure d'un acteur peut se composer de références d'acteurs, de ports finaux, de liaisons et d'équivalences. Ces éléments sont illustrés à la figure 1. Les références d'acteurs sont représentées par les boîtes blanches (*actor2R1, actor3R1*). Les ports sont représentés par les carrés noirs situés sur les limites de l'acteur (*protocol1R1, protocol1R2* et *protocol2R1*). Les liaisons sont les lignes qui joignent les ports.

Les références d'acteurs sont les incarnations des classes d'acteurs. Il ne peut y avoir aucun partage de donnée entre un acteur et les acteurs qu'il contient. Ils doivent communiquer par l'échange de messages par les ports. Ces derniers, qui sont des références à des protocoles (dont il sera question plus loin), sont appelés ports finaux lorsque définis à l'intérieur d'un acteur. Ils sont utilisés pour communiquer avec des acteurs contenus ou avec les services du système tel l'horloge ou l'intercepteur d'anomalies. Les ports doivent

être reliés pour pouvoir communiquer. Un lien explicite, appelé liaison, doit être defini entre deux ports (ou entre un port et un point d'accès de service). Les conditions qui régissent les liaisons seront présentées dans la section consacrée aux communications. Le modèle ROOM permet l'appartenance multiple, c'est-à-dire qu'un même acteur peut appartenir à plusieurs acteurs en même temps. Cette situation est spécifiée à l'aide des équivalences. Toutes les chaînes d'acteurs menant à une même référence sont énumérées. Ceci permet à une classe d'acteurs de considérer plusieurs références comme une seule incarnation.

Une classe d'acteurs possède des propriétés de généricité qui sont spécifiées lors de sa définition: ses incarnations peuvent être remplacées par celles d'une autre classe, elles peuvent se substituer à celles d'une autre classe, ou aucun de ces deux cas. Pour qu'une substitution soit possible, les interfaces des deux classes impliquées doivent être compatibles. C'est-à-dire que la classe substituante doit avoir au moins les mêmes ports que la classe substituée. La substitution est possible lors de la création dynamique d'acteurs pendant l'exécution du modèle et non lors de la création statique initiale.

Les références d'acteurs

Étant donné qu'une classe d'acteurs n'est qu'une définition de type, elle doit être incarnée. Dans le modèle ROOM, ces incarnations sont accessibles par des références. Une référence d'acteur fournit la majeure partie des informations concernant une incarnation comme le nom de sa classe, le facteur de reproduction, la méthode d'incarnation et si la substitution par une autre classe est possible. Une référence peut pointer à une ou plusieurs incarnations auquel cas un facteur de reproduction est utilisé pour en indiquer le nombre. Ces dernières sont alors stockées dans un vecteur. Une seule référence peut donc permettre l'accès à plusieurs incarnations d'une même classe. Un acteur est incarné lors de la création statique du modèle, ou dynamiquement lors de son exécution. La référence peut aussi être déclarée comme un endroit réservé qui pourra être incarné

dynamiquement par un acteur compatible. Une référence doit être spécifiée de façon explicite comme étant remplaçable pour qu'elle soit générique.

Les objets de données

Dans ROOM, chaque parcelle d'information doit faire partie d'une classe. Cette exigence amène les avantages du paradigme orienté objet [2] à la partie donnée du modèle. Un modèle fait appel à une certaine quantité de classes de données prédéfinies. De nouvelles classes peuvent également être spécifiées dans le langage détaillé qui peut être soit le Real-Time Programming Language (RPL) ou C++. Une classe de données spécifie le type de l'information et les méthodes qui en permettent l'accès et la manipulation.

1.1.2 Le modèle de communication

Tel que mentionné, tous les acteurs dans ROOM sont encapsulés et ne partagent rien entre eux. Ils offrent seulement des services qui peuvent être requis par des canaux de communication convenablement établis. Chaque requête de service correspond à un message spécifique. Le modèle de communication est maintenant présenté en deux sous-sections, les classes de protocoles et les ports.

Les classes de protocoles

L'ensemble des messages pouvant être échangés entre deux acteurs est appelé un protocole. Son type est spécifié par une définition de classe. L'héritage est disponible dans les classes de protocoles. Le service de communication simulée peut être utilisé afin d'introduire un délai artificiel dans la transmission de messages. Ce service permet de retarder la réception des messages d'un protocole. Chaque message est défini par une direction, une identification et le type d'information qu'il transporte. La direction indique si le message sera reçu (direction *entrante*) ou envoyé (direction *sortante*). L'importance

de la direction sera expliquée plus en détail ultérieurement. L'identification est également appelée *signal* et est donnée sous forme d'un nom. Lorsqu'un message est reçu, un acteur peut, à partir du *signal*, déterminer si des données sont incluses et, le cas échéant, de quel type il s'agit. Un message transporte l'information sous forme d'objets de données. Lorsqu'aucune information n'est incluse, le message contient l'objet *nul*.

Les ports

L'incarnation d'une classe de protocoles est un port. Dans un acteur, les ports servent à envoyer et à recevoir des messages. Tel que mentionné précédemment, chaque message circule dans une direction (*entrante* ou *sortante*). Afin de permettre la communication entre deux ports, l'un d'eux doit être conjugué. Par conjugaison, il est entendu que les directions des messages du protocole sont inversées. Autrement dit, les messages sortants deviennent entrants et inversement. Les ports de différents acteurs doivent être liés pour pouvoir communiquer. Lorsqu'un acteur doit se lier à un acteur reproduit, son port et la liaison doivent également être reproduits par le même facteur. L'échange de messages ne se limite pas à un acteur et aux acteurs qu'il contient. Une liaison peut traverser plusieurs interfaces pour atteindre un acteur qui appartient à un autre propriétaire. Pour que ce type de liaison soit conforme au modèle, la traversée d'interfaces doit se faire par le biais de ports de relais. Ces derniers ne peuvent envoyer ou recevoir des messages, ils servent exclusivement au franchissement d'une interface. Ils ne modifient les messages d'aucune façon.

1.1.3 La modélisation comportementale

La partie dynamique du modèle est définie dans la spécification du comportement. Le comportement d'un acteur est décrit de deux façons. La première est le code d'implantation qui sera exécuté à la mise en marche du système. La deuxième est une machine

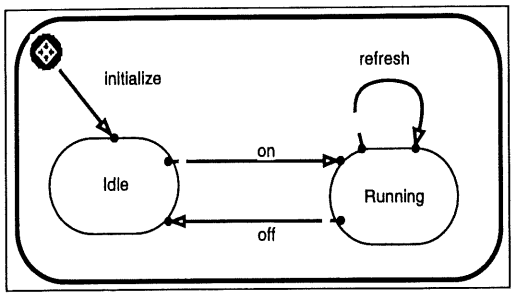

FIG. 2 – *ROOMchart ou diagramme de comportement ROOM*

à états finie. Dans l'environnement ROOM, les machines à états sont appelées ROOMcharts et sont basées sur le formalisme Statecharts de Harel [9], [10]. L'implantation et le formalisme des ROOMcharts sont maintenant abordés plus en détail.

Le langage détaillé

Tout le code lié au comportement d'un acteur doit être exprimé dans le même langage. Jusqu'à maintenant, ROOM permet deux langages: Real-Time Programming Language (RPL) et C++ [27]. RPL est un langage dérivé de Smalltalk-80 [7] adapté au modèle ROOM. Il a été conçu pour permettre un prototypage rapide mais est cependant limité à la simulation. Lorsqu'un modèle exécutable est créé pour une plate-forme cible, le code doit être écrit en C++. Les fonctions qui sont appelées à plusieurs endroits dans un ROOMchart sont déclarées explicitement une seule fois. Lorsque C++ est utilisé, les fichiers d'inclusion nécessaires doivent être déclarés dans le comportement de l'acteur. Le code exprimé en langage détaillé peut se retrouver sous forme d'action lors d'une transition, de condition de garde dans une spécification de déclenchement, d'action à l'entrée ou à la sortie d'un état, d'une fonction interne ou d'une condition sur une fourche d'alternatives.

Les ROOMcharts

Un ROOMchart est une machine à états finie hiérarchique. Les états peuvent se décomposer en sous-états pour permettre l'abstraction. Il en résulte une conception comportementale plus compacte et plus lisible. Un ROOMchart est défini par un état racine qui peut contenir des variables, du code d'entrée ou de sortie, un ensemble de sous-états, un ensemble de transitions et un ensemble de fourches d'alternatives. La représentation graphique d'un ROOMchart est illustrée à la figure 2. Les états sont représentés par des rectangles aux coins arrondis et les transitions par les flèches qui les relient. Le cercle noir dans le coin supérieur gauche du diagramme symbolise l'état initial dans lequel se trouve l'acteur immédiatement après sa création. La transition qui quitte l'état initial est franchie automatiquement sans attendre un déclencheur (réception d'un message). Une déclaration de variable varie selon le langage détaillé utilisé mais demeure dans tous les cas une référence à une classe de données. Les définitions de code d'entrée ou de sortie sont des séquences d'énoncés qui seront exécutés lorsque le contrôle entre ou sort d'un état. L'ensemble des sous-états contient les spécifications d'états ainsi que les noms respectifs qui les identifient.

Les spécifications de transitions donnent la liste des transitions d'un ROOMchart. Ces dernières permettent au contrôle de passer d'un état à un autre. Une transition peut être franchie lors de la réception de messages spécifiques. Chaque transition est déclenchée par un signal. L'échange de messages est donc le principal catalyseur de l'exécution du modèle. Lorsque l'acteur est dans un état et qu'il reçoit un message, il vérifie si une transition peut être déclenchée par le signal reçu. Si oui, le processus de transition démarre, sinon le message est abandonné. Des conditions de garde peuvent être définies afin de restreindre le franchissement de transitions. Ces conditions sont en fait des fonctions qui retournent des valeurs booléennes. Pour que la transition soit franchie, la fonction doit retourner la valeur *vrai*. Des actions peuvent être attachées à une transition. Elles sont exécutées au moment où la transition est déclenchée. Le code du langage détaillé mentionné plus haut

est exécuté seulement lors d'une transition. L'ordre dans lequel ce code est évalué est le suivant: la condition de garde, le code de sortie de l'état quitté, le code de transition et le code d'entrée de l'état atteint. Toutes les possibilités mentionnées précédemment sont facultatives.

Une fois qu'une transition est déclenchée, elle ne peut être redirigée vers un autre état. Il est utile de pouvoir changer la destination d'une transition suite à un traitement quelconque. Le concept des fourches d'alternatives a été créé afin de permettre cette possibilité. Les fourches d'alternatives sont en quelque sorte des états intermédiaires où une fonction booléenne est évaluée. Seulement deux transitions peuvent quitter une fourche d'alternatives : une branche pour le cas où la fonction retourne *vrai* et une autre pour le cas où elle retourne *faux*. Les transitions peuvent mener à une fourche d'alternatives ou en quitter une. De cette façon, il est possible d'exécuter du code avant et après une fourche.

1.2 ROOM et ses pairs

L'utilisation dans l'industrie de méthodologies reconnues [2], [25], [3], [4], [11], [32], [6], est maintenant chose commune. Cependant, ces dernières ont, en général été créées pour le développement de systèmes d'information. Malheureusement, leur généricité les rend souvent inappropriées lors du développement d'applications à caractère particulier tel le domaine des systèmes temps-réel. Particulièrement dans ce domaine, la modélisation ou prototypage est souvent préférée à l'approche du développement en cascade [24]. La majorité des outils qui soutiennent les méthodologies génériques n'offrent qu'une vérification de la sémantique statique d'un modèle et non celle de la sémantique dynamique. Une méthodologie et un langage adaptés au développement de systèmes temps-réel, l'Object-Oriented Real-Time Techniques (OORT) [20] et la Unified Method Language for Real-Time Systems Design (UMLRT) [28], [29], seront maintenant brièvement présentés.

1.2.1 OORT

OORT est une méthodologie dédiée aux systèmes temps-réel qui comprend les activités suivantes : l'analyse des requis, la conception architecturale, la conception détaillée, la conception des tests, l'implantation et la phase de test. Cette méthodologie suit un processus itératif où toutes les activités sont faites en séquence sur chaque partie du système. Pour illustrer les résultats des activités du processus, OORT utilise certaines parties de la notation d'OMT [25], le *Specification and Description Language* (SDL) [30] et les *Message Sequence Charts* (MSC) [16]. Comme des outils très distincts sont utilisés pour décrire le développement d'un système, des règles de cohérence ont été définies afin d'éviter les erreurs entre les différentes représentations. L'outil ObjectGEODE [19] de Verilog implante cette méthodologie et ces notations. Les activités du processus de développement ainsi que les notations qui leur sont associées seront maintenant abordées plus en détail.

OORT recommande l'utilisation de la modélisation objet et les MSCs pour documenter l'analyse des requis. Les informations requises par l'utilisateur sont illustrées par les diagrammes d'instances et de classes d'OMT. Tous les objets pertinents au domaine de l'application devraient y être représentés. Ces derniers peuvent être de nature logique ou physique et peuvent provenir du système ou de son environnement. La notation MSC est utilisée pour identifier et organiser les fonctions attendues du système et pour décrire les scénarios finaux. Afin de s'assurer de la cohérence entre ces diagrammes, les objets des MSCs devraient être des instances des classes OMT. Les messages des MSCs doivent être des propriétés des classes et les transmissions de messages doivent être rendues possibles grâce à des liens associatifs OMT entre l'émetteur et le récepteur.

Les résultats de la conception architecturale sont représentés à l'aide du langage SDL. L'architecture du système est illustrée par un diagramme hiérarchique et un ensemble correspondant de diagrammes d'interconnexions. La conception des tests, qui est faite

parallèlement à la conception architecturale en utilisant des MSC, aide à raffiner l'architecture.

La conception détaillée utilise les diagrammes de processus SDL pour décrire les objets concurrents terminaux. De tels objets sont les feuilles du diagramme hiérarchique SDL. Les objets passifs sont représentés à l'aide des diagrammes de classe OMT. Ces objets passifs proviennent de classes obtenues lors de l'analyse des requis et de types abstraits de données obtenus de l'architecture du système. Les MSC de la conception architecturale sont raffinés durant cette activité afin de compléter les tests d'intégration et de produire les tests unitaires.

La prochaine activité est l'implantation. Elle consiste à transformer les spécifications de la conception détaillée en code qui s'exécutera sur le système d'exploitation choisi. Cette tâche consiste à prendre les spécifications SDL et les classes des diagrammes OMT, et à les transposer dans le langage d'implantation. Les objets passifs des classes des diagrammes OMT sont souvent des squelettes de code et elle doivent être complétées manuellement.

La phase de test prend les MSC qui proviennent de la conception détaillée et vérifie si l'exécution est correcte. Afin d'augmenter l'efficacité de la phase de test, cette dernière devrait être faite dans l'environnement cible.

1.2.2 UML pour la conception de systèmes temps-réel

UMLRT [28], [29] est, actuellement, un langage de modélisation non accompagné d'une méthodologie. Il est prévu que les auteurs Booch et Rumbaugh adaptent respectivement leur méthode à UMLRT. La notation de UMLRT se compose des types de diagrammes suivants: de classes, d'utilisation (*use-case*), d'interactions, d'états, de composantes et de déploiement.

Les diagrammes de classes sont au coeur du modèle UMLRT. Ils illustrent les principales abstractions du système et la manière dont elles sont reliées. Ils se composent

de deux types d'éléments graphiques soient les symboles de classes et les symboles de relations. Les premiers représentent les classes, leurs attributs et leurs opérations. Les seconds, illustrent les associations, les aggrégations, l'héritage, les dépendances et les instanciations.

Les diagrammes d'utilisation donnent un aperçu général de la façon dont le système sera utilisé. Ils offrent une vue statique de haut niveau des fonctions offertes. Ils sont utiles lors de la spécification des requis.

Les diagrammes d'interactions modélisent l'aspect dynamique des systèmes. Ils mettent en relief des notions de séquence et de temporisation. UMLRT propose deux manières de décrire les interactions: les diagrammes de séquence et les diagrammes de collaborations. Les premiers illustrent les objets ainsi que les messages qu'ils échangent entre eux de façon séquentielle. Les diagrammes de collaborations montrent l'échange des messages et la disposition structurelle des composantes impliquées.

Le côté dynamique d'un système est aussi illustré à l'aide de diagrammes d'états. Ceux proposés dans UMLRT sont dérivés des *statecharts* de Harel. Ils permettent de spécifier les réactions du système aux événements qui surviennent.

Les diagrammes de composantes décrivent les éléments de l'implantation qui correspondent aux entités logiques de la modélisation. En C++, par exemple, une classe peut s'implanter à l'aide de deux fichiers, un de type ".h" pour la définition de la classe et un de type ".cpp" pour la définition des méthodes.

Les diagrammes de déploiement indiquent les associations entre les parties matérielles du système et les composantes logicielles auxquelles elles sont liées. Il en résulte donc une vue globale du système implanté avec tous les éléments impliqués.

Chapitre 2

La modélisation orientée objet d'un logiciel de poursuite de satellites

Ce chapitre[1] détaille la conception selon le modèle ROOM et l'implantation d'un logiciel de poursuite en temps réel de satellites. L'application résultante n'est pas un simple exemple académique et est présentement utilisée dans une station de communications par satellites. La principale contribution de ce travail est l'utilisation d'une technique de modélisation de systèmes temps-réel dans le développement d'une application concrète. L'implantation du modèle mathématique de l'orbite est le fruit de la contribution de M. Barbeau.

2.1 Introduction au domaine du problème

Plusieurs centaines de satellites tournent en orbite autour de la terre. Parmi ceux-ci, certains sont utilisés pour la météorologie, pour la retransmission de signaux de télévision

1. Le contenu de ce chapitre est tiré d'une publication intitulée "*Object-Oriented Modeling of a Satellite Tracking Software*" [17]. Cette dernière a été présentée à la *15th ARRL and TAPR Digital Communications Conference* à Seattle en septembre 1996 où elle a été primée meilleur article à caractère technique et théorique par un étudiant.

ou de téléphonie, pour l'espionnage et pour la recherche scientifique. La station orbitale russe Mir et les navettes spatiales américaines sont également considérées comme des satellites. Tous les signaux qu'ils émettent sont recevables mais pas nécessairement déchiffrables. La majorité d'entre eux ne permettent pas la réception de signaux émis par le public. Afin de pallier à cette situation, la communauté radio amateur a construit et mis en orbite ses propres satellites pour permettre la retransmission de signaux radios qui lui sont réservés [5]. Présentement, une cinquantaine de satellites radio amateurs, la station Mir et les navettes spatiales ont tous la capacité de recevoir et de transmettre des signaux radio sur des fréquences accessibles au public. Une des caractéristiques communes des satellites est le fait que leur orbite n'est pas géostationnaire. Ceci imlique l'utilisation d'antennes directionnelles pointées vers le satellite lorsque celui-ci est visible pour obtenir une bonne communication. Le satellite doit se trouver dans le champ visuel de l'observateur afin que la transmission et la réception de signaux soient possibles. La direction des antennes doit constamment changer pour suivre la trajectoire du satellite dans le ciel.

Les orbites sont circulaires ou elliptiques (voir figure 3). Comme un satellite doit être visible pour être accessible, le temps qu'il prend à passer d'un point de l'horizon à un autre s'appele une passe. La ligne, à la figure 3, qui traverse les deux orbites délimite une passe pour la station terrestre (illustrée par le triangle). Dans le cas de la station orbitale Mir, qui décrit une orbite circulaire, la durée d'une passe est approximativement 12 minutes. Oscar 10, qui suit une trajectoire elliptique, peut prendre jusqu'à 8 heures pour compléter une passe.

Comme les orbites circulaires évoluent toujours à la même altitude, l'intensité du signal émis est approximativement constante (si les distorsions sporadiques sont ignorées). Dans le cas d'une orbite elliptique, la force du signal s'atténue quand le satellite s'éloigne de la terre et s'intensifie lorsqu'il revient. La faiblesse du signal alors que le satellite est à son apogée accentue le rôle du caractère directionnel. Toutefois, certains satellites tels

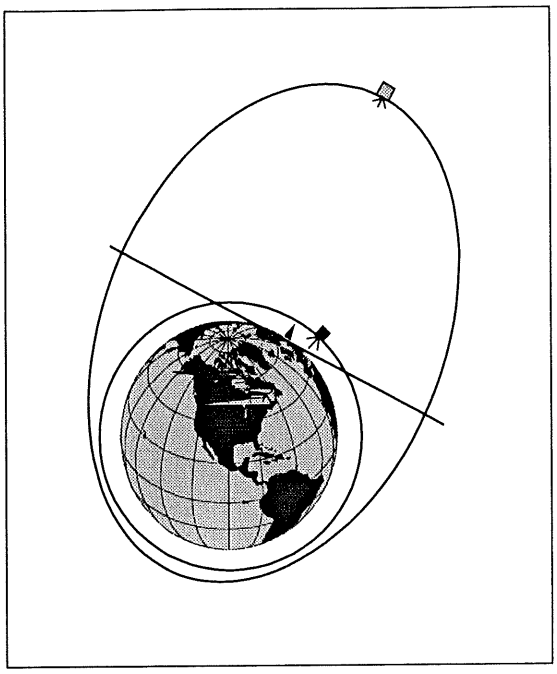

FIG. 3 – *Modèles orbitaux*

Mir diffusent un signal assez fort pour être capté à l'aide d'antennes omnidirectionnelles.

2.2 Analyse

Le problème à résoudre est le développement d'un logiciel de poursuite en temps réel de satellites non géostationnaires (en orbite circulaire) [5]. L'application qui en résulte se nomme SatSy. Poursuivre un satellite signifie déterminer sa position dans l'espace et vers où les antennes doivent être pointées. En termes fonctionnels, l'utilisateur de SatSy peut faire les actions suivantes : sélectionner un satellite, obtenir la direction des antennes, démarrer ou arrêter la poursuite. Ces opérations sont rendues disponibles par le biais d'une interface graphique. Le problème de la poursuite est composé de deux parties, le calcul de la position du satellite en coordonnées terrestres et le calcul de l'azimut et de l'élévation du satellite par rapport à un endroit au sol.

2.2.1 Le suivi terrestre

La première partie de ce problème se nomme le *suivi terrestre.* Il faut calculer la position dans l'espace d'un satellite à un moment donné. Cette position est rapportée en utilisant des coordonnées terrestres: la longitude, la latitude et l'altitude. Les deux premières forment le point sous-satellite (*PSS*). Le PSS représente le point à la surface de la terre qui est directement sous le satellite. Pour simplifier le texte, le PSS réfère à la combinaison du point à la surface et de l'altitude.

Les paramètres de ce problème sont le temps sidéral moyen de Greenwich (TSMG) au 00.00Z de chaque année et l'ensemble des éléments orbitaux d'un satellite. Le TSMG est calculé à partir du temps fourni par le système d'exploitation. Les éléments orbitaux sont disponibles sous forme d'un fichier textuel qui peut être obtenu d'un serveur Internet du *United States Air Force Institute of Technology.* Ce fichier fournit de l'information sur la position des satellites observés à des moments précis. Il est disponible sous deux formats différents soient: *NASA* et *AMSAT.* Le format *NASA* est plus compact alors que le format *AMSAT* est plus facile à lire. Le premier format a été retenu pour cette application à cause de sa taille réduite et dû fait qu'il est syntaxiquement plus facile à analyser. Chaque enregistrement du fichier qualifie la position d'un satellite. Comme ces derniers sont sensibles aux perturbations telles l'attraction terrestre, ces données sont pertinentes pour une courte période de temps (ex: une semaine pour les satellites à basse altitude).

2.2.2 Calcul de la direction

La direction où les antennes doivent pointer se calcule à partir du PSS obtenu au suivi terrestre. La direction est exprimée par deux termes: l'azimut et l'élévation. Le permier représente un angle horizontal compris entre 0 et 360 degré(s). À 0 (ou 360) degré, l'antenne pointe vers le nord, à 90 vers l'est, à 180 vers le sud et à 270 vers

l'ouest. L'élévation est l'angle vertical par rapport au sol. Il est compris entre 0 et 180 degrés. À ces deux valeurs, il pointe vers l'horizon. Lors d'une passe de satellite, une élévation haute (qui s'approche de 90 degrés) donne une bonne qualité de signal et un temps d'accès plus long. Les paramètres requis sont le PSS, la longitude et la latitude de la station terrestre. Les coordonnées terrestres de la station sont nécessaires, car la direction des antennes leur est relative. L'altitude de la station pourrait être considérée, mais, selon Davidoff [5], son influence est négligeable. Le choix de la plate-forme sur laquelle l'application est implantée sera maintenant justifié.

2.2.3 La plate-forme logicielle

La dimension temps-réel de cette application impose le choix d'un système d'exploitation adapté à ce domaine. La plate-forme cible retenue est QNX. Les raisons qui justifient ce choix sont: la taille de son micro-noyau (15 K octets) et sa compatibilité avec la famille de processeurs Intel 80x86. La taille très réduite de son micro-noyau permet à celui-ci de rester dans l'antémémoire du processeur, le rendant ainsi très performant. Les inconvénients de ce choix sont le fait que les *threads* et la communication interprocessus asynchrone ne soient pas disponibles.

2.3 L'architecture et la conception

2.3.1 Version avec interface en mode texte

Un modèle préliminaire [17] a mené à trois principaux acteurs: *console*, *controller* et *tracker*. L'acteur *console* joue un rôle d'interface entre l'utilisateur et le système. Il est responsable d'offrir les actions possibles à l'utilisateur, d'obtenir ses entrées et de les envoyer à *controller*. La structure de *console* ne contient que deux ports. Un pour communiquer avec l'utilisateur et un pour communiquer avec *controller*. Les actions disponibles

à l'utilisateur sont : initialiser une poursuite, démarrer une poursuite, arrêter une poursuite et quitter. Lors de l'initialisation d'une poursuite, l'utilisateur doit sélectionner un satellite. L'ensemble des actions possibles dépend de l'état de *console*. Par exemple, si aucune poursuite n'est engagée, il est impossible d'en arrêter une. Cette interface n'est pas graphique et fournit un minimum d'information à l'utilisateur.

Le deuxième acteur, *controller*, exécute les actions demandées par l'utilisateur au travers de *console*. Une fois la poursuite engagée, il est responsable de fournir à *tracker* la direction dans laquelle les antennes doivent être pointées. L'acteur *controller* contient deux autres acteurs soient: *groundTracker* et *azElProvider*. Ses fonctions principales sont d'initialiser les paramètres de la station, d'initialiser une poursuite et de calculer l'azimut et l'élévation des antennes à un intervalle de temps spécifique. L'acteur *controller* a un port permettant la communication avec *console* et un autre port pour l'envoi de messages à *tracker*.

Le dernier acteur au niveau supérieur est *tracker*. Il joue le rôle de pilote de la carte d'interface du rotor. Il reçoit l'azimut et l'élévation de *controller*. Il envoie alors ces coordonnées à l'interface du rotor par des fonctions de bas niveau. Le flux des messages passe de l'utilisateur à *console*, de *console* à *controller* et finalement, de *controller* à *tracker*. Cette architecture mène à un couplage très faible et une forte cohésion des acteurs. Par contre, le manque d'information fournie à l'utilisateur a forcé quelques changements.

2.3.2 Version avec interface en mode graphique

Nous avons étendu notre logiciel avec une interface graphique. Celle-ci a conduit à un accroissement substantiel du nombre de messages échangés parce qu'un modèle graphique de la course du satellite est présenté à l'écran. L'augmentation des flux de messages a forcé certaines modifications au modèle. Dans la version précédente, toute l'information passait par *controller*. Les données obtenues de *groundTracker*, *azElProvider* et *tracker* n'étaient pas acheminées à *console* pour être affichées. Pour pouvoir réaliser ceci, tous les

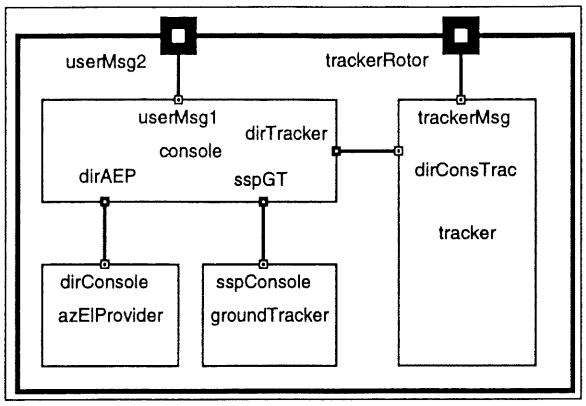

FIG. 4 – *La structure de SatSy*

messages se dirigeant vers *console* auraient passé par un acteur intermédiaire (*controller*) avant d'atteindre leur destination. La logique de *controller* a donc été incluse dans *console*. Cette nouvelle architecture (voir figure 4) force maintenant *groundTracker* et *azElProvider* à fournir leurs services directement à *console*. L'acteur *tracker* communique également d'une façon directe avec *console*.

Le comportement de *console* est illustré à la figure 5. La transition (*getOrbitalElements*) entre l'état initial (le cercle noir) à l'état *Idle* effectue le travail de lecture de la longitude et de la latitude de la station terrestre et transmet cette information à *azElProvider*. La transition (soit de *Idle* ou *Ready* à *Ready*) étiquetée *SatelliteChoice* est déclenchée par le message *SetElements* qui provient de *console* et contient les éléments orbitaux d'un satellite choisi. La transition étiquetée *TrackingSwitch*, de *Ready* à *Tracking*, est déclenchée par le message *StartTracking* et elle démarre la poursuite. La transition réflexive *timeout* est déclenchée par une minuterie et est responsable de calculer la nouvelle direction des antennes. La transition *TrackingSwitch* de *Tracking* à *Ready* arrête la poursuite et survient lors de la réception du message *StopTracking*. Les deux transitions *OrbitalTrack* sont déclenchées par le message *OrbitalTrack* qui indique que le tracé orbital doit être affiché ou effacé. Ces dernières mènent à des fourches d'alternatives qui déterminent si le tracé est affiché ou non. Selon le cas, la transition appropriée sera

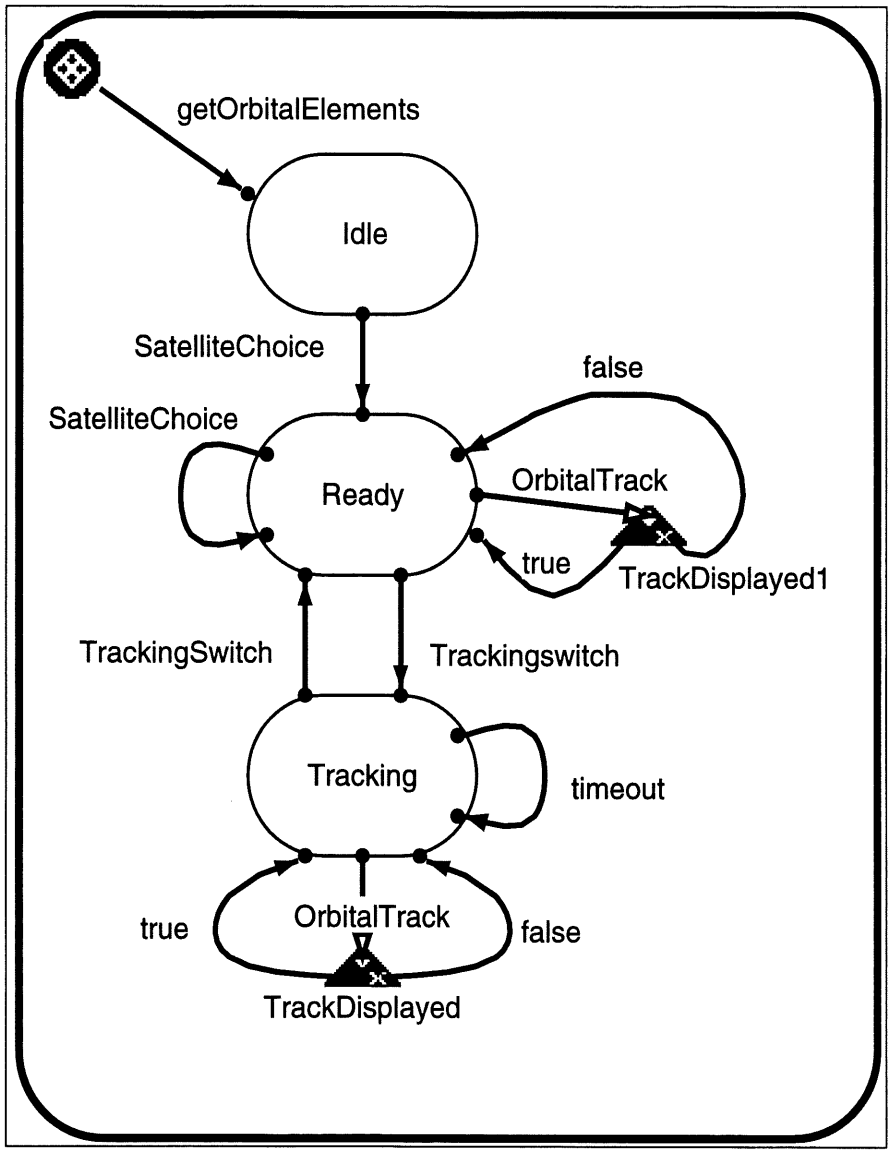

FIG. 5 – *Le comportement de* console

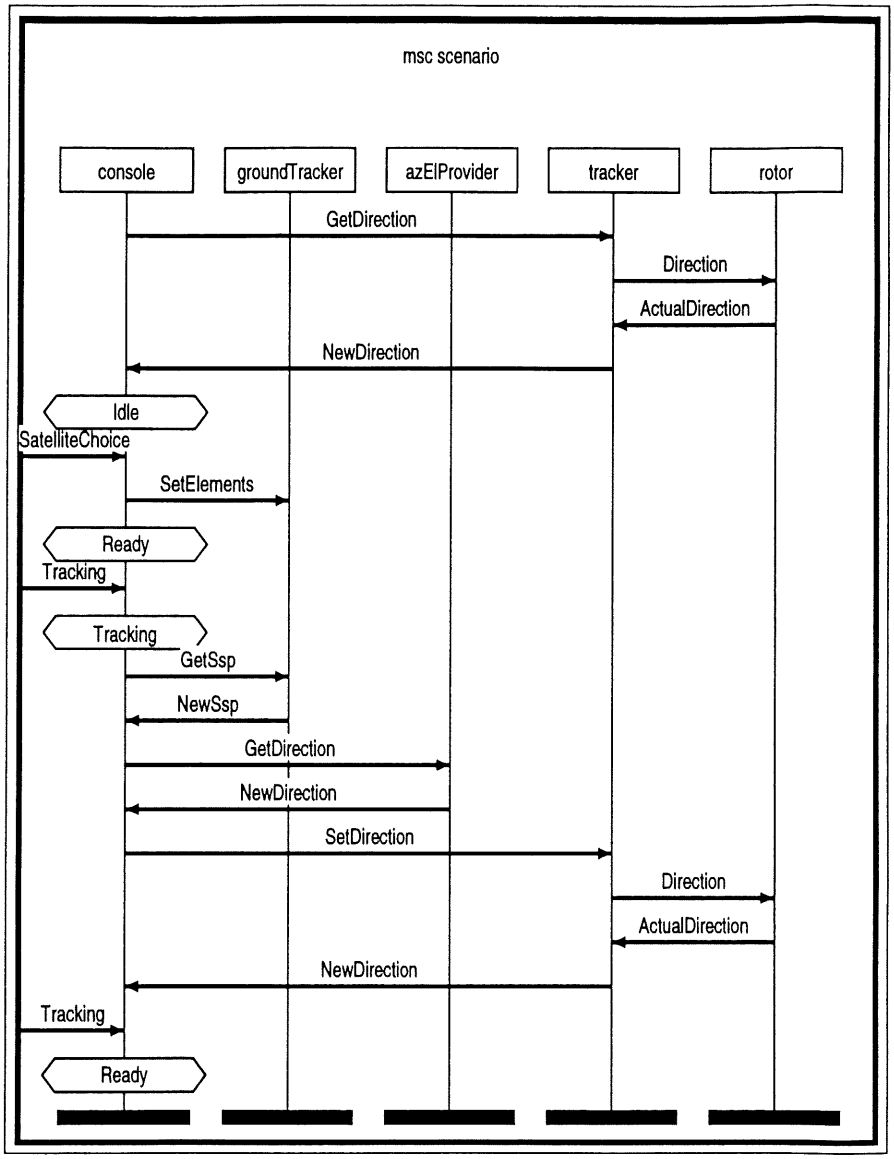

FIG. 6 – *Le scénario typique d'une poursuite*

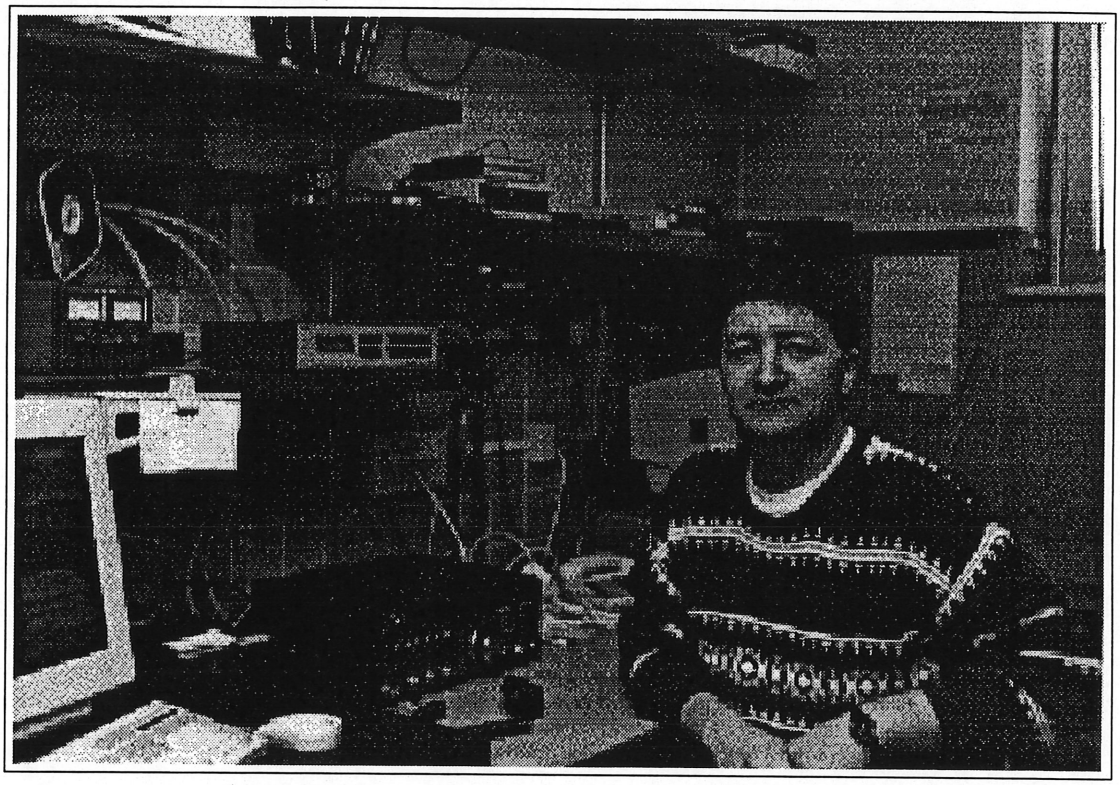

FIG. 7 – *L'installation radio d'une station de communications par satellites*

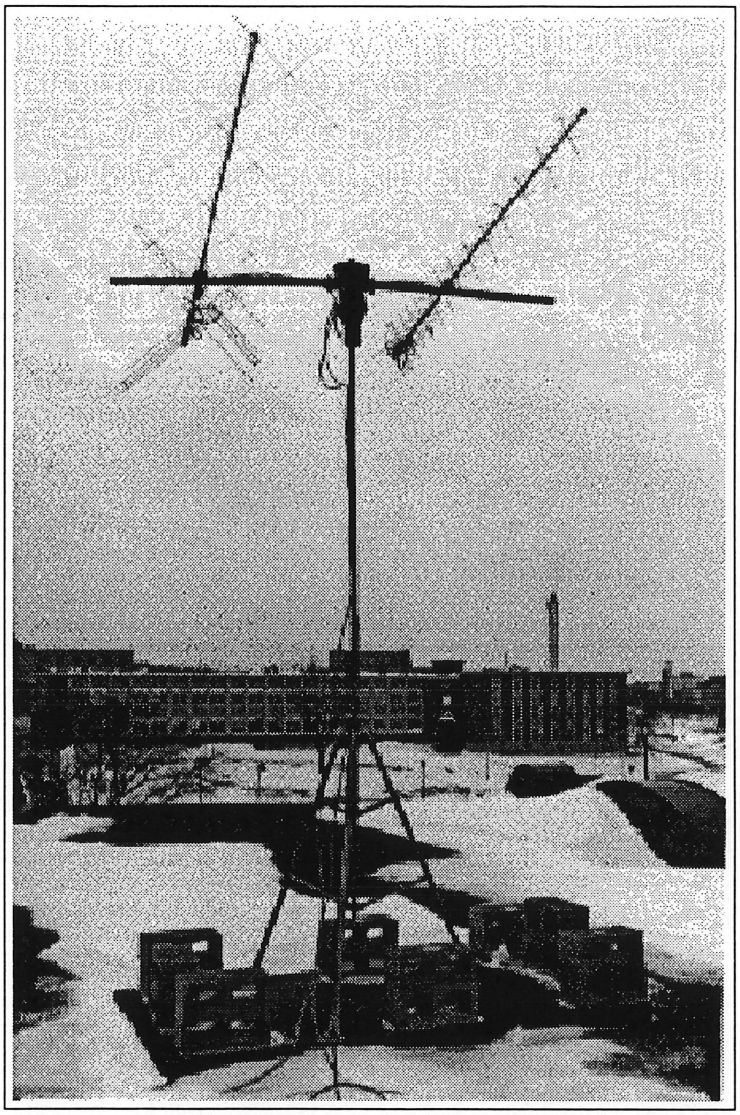

FIG. 8 – *Les antennes et le rotor d'une station de communications par satellites*

franchie.

Le déroulement typique d'une séance de poursuite est illustré à la figure 6. Lorsque le système démarre, *console* envoie un message à *tracker* pour obtenir la direction dans laquelle les antennes pointent. Une conversion est alors démarrée par *tracker* pour recevoir la direction sous un format numérique. Cette dernière est retournée à *console* par *tracker*. L'acteur *console* attend alors une sélection de satellite de la part de l'utilisateur. Lorsque *console* la reçoit, il envoie les éléments orbitaux correspondants au choix fait à *groundTracker*. SatSy est maintenant prêt pour la poursuite et attend une commande de l'utilisateur. La commande *Tracking* reçue par *console* démarre la boucle de poursuite. Cette boucle s'exécutera à un intervalle de temps fixe jusqu'à ce que la poursuite soit interrompue. Ce cycle commence par la requête *GetSSP* faite par *console* à *groundTracker*. Lorsque *console* reçoit le message *NewSSP* de *groundTracker*, il l'inclut dans la requête *GetDirection* faite à *azElProvider*. Le message *NewDirection* de *azElProvider* à *console* contient la direction dans laquelle les antennes doivent pointer. L'acteur *console* envoie alors le message *SetDirection* qui contient la nouvelle direction à *tracker*. Sur réception de ce message, *tracker* répond à *console* avec la direction réelle des antennes. L'acteur *tracker* se charge alors de positionner les antennes. La réception par *console* d'un deuxième message *Tracking* signifie que la poursuite doit être interrompue. À ce point, l'utilisateur peut choisir un autre satellite ou redémarrer la poursuite.

2.4 Implantation

Dans cette section, nous discuterons de l'implantation du logiciel. Les points importants sont le matériel et le système d'exploitation sur lequel l'application s'exécutera, le langage de programmation utilisé et la correspondance établie entre les concepts de notre modèle et les concepts du langage choisi. Considérons premièrement le matériel.

Nous avons choisi d'utiliser un micro ordinateur de type Pentium, un émetteur récepteur THF/UHF, un processeur à signaux numériques (illustrés à la figure 7), un rotor d'azimut et d'élévation avec sa carte d'interface, une antenne THF et une antenne UHF (illustrés à la figure 8).

Nous avons choisi le système d'exploitation QNX qui est spécialement adapté aux applications temps-réel. Il est multi-tâche et implante le changement de contexte rapide. QNX possède une architecture micro-noyau ce qui signifie que son noyau est minimal et peut demeurer dans l'antémémoire du processeur. Il est donc très performant. Plusieurs processus peuvent s'exécuter de façon concurrente et QNX fournit des primitives de communication interprocessus par messages (*send, receive, reply*). Le langage de programmation retenu pour l'implantation est C++ car, parmi ceux disponibles sur QNX, il est conceptuellement le plus près du modèle ROOM.

Les acteurs dans le modèle ROOM correspondent soit à des processus QNX concurrents, soit à des objets C++. Il est à noter que dans notre modèle ROOM, certains acteurs peuvent s'exécuter en parallèle (tels que *console*, *controller* et *tracker*) alors que certains autres s'exécutent en séquence (comme *controller*, *groundTracker* et *azElProvider*). D'un côté, nous implantons les acteurs qui s'exécutent en parallèle comme des processus concurrents qui communiquent par échanges de messages. D'un autre côté, les groupes d'acteurs qui s'exécutent en séquence sont implantés dans le même processus. Chaque acteur devient un objet et communique avec les autres objets par des appels de fonctions C++. Ceci permet d'éviter le surplus de travail associé aux primitives de communication interprocessus occasionnant un gain d'efficacité. L'ensemble des acteurs regroupant *controller*, *groundTracker* et *azElProvider* est donc implanté dans un seul processus alors que les acteurs *console* et *tracker* seront les seuls objets de leur processus. Il résulte donc trois processus de notre conception. Cette stratégie d'implantation offre une organisation de processus simple et efficace.

La communication interprocessus asynchrone a été utilisée dans notre implantation

même si QNX n'offre pas les primitives nécéssaires. Un tel mode de communication a été simulé avec l'aide des signaux. Lorsqu'un processus désire envoyer un message, il lève un signal chez le processus ciblé. Le traitement de l'interruption engendrée met alors le processus en mode réception synchrone. L'échange de messages peut maintenant avoir lieu. Cette façon de procéder évite à un processus de bloquer sur l'attente d'une réception de message. Le traitement peut donc se poursuivre entre chaque échange.

2.5 Discussions et perspectives

Le logiciel résultant de cette étude de cas diffère des autres logiciels de poursuite de satellites par sa conception. Certains logiciels offrent plus de possibilités que SatSy. Cependant, l'architecture orientée objet de notre logiciel le rend facilement extensible. En effet, l'introduction d'une interface graphique n'a affecté qu'un seul acteur (*console*) et en a éliminé un autre (*controller*). Cette conception a aussi mené à des acteurs qui sont réutilisables. Un nouveau projet qui offrira des accès multiples à une station de communications par satellites réutilise déjà la majorité des acteurs de SatSy.

Chapitre 3

La sémantique statique du langage ROOM

La formalisation de la sémantique statique du langage ROOM proposée dans ce chapitre est faite à l'aide des grammaires attribuées. ROOM utilise deux types de langages, soit un langage de haut niveau et un autre de niveau détaillé. Le langage de haut niveau décrit le modèle ROOM alors que le deuxième porte sur le comportement de bas niveau. Ce dernier peut être C++ ou RPL. La formalisation présentée dans ce chapitre traite uniquement le langage de haut niveau. Un bref aperçu des grammaires attribuées est présentée dans la suite de ce chapitre. La grammaire attribuée du langage ROOM que nous avons développée est détaillée à l'annexe A.

La formalisation de la sémantique statique du langage ROOM requiert une technique simple et adaptée à notre besoin. Le résultat de l'analyse d'un modèle ROOM sous forme linéaire est représenté par des structures arborescentes correspondant aux règles syntaxiques du langage. Chaque noeud de l'arbre est associé à une règle syntaxique et est enrichi par des attributs qui décrivent ses propriétés. À l'analyse, lorsqu'une règle est applicable, un noeud est créé et les valeurs de ses attributs sont extraites de l'environnement du moment. Les règles d'attribution peuvent être spécifiées par le formalisme des

grammaires attribuées. Il couvre la définition formelle des propriétés contextuelles et non contextuelles des langages.

Une grammaire attribuée est définie par le quadruplet $AG = (G, A, R, B)$ où G est une grammaire non contextuelle, A un ensemble fini d'attributs, R un ensemble fini de règles d'attribution et B un ensemble fini de conditions. Les attributs sont soit synthétisés ou soit hérités. Dans le premier cas, les valeurs sont déterminées à partir des attributs des noeuds qui sont hiérarchiquement plus bas dans l'arborescence. Dans l'autre cas, les valeurs sont déterminées à partir des attributs des noeuds qui sont hiérarchiquement plus haut. La somme des attributs hérités est appelée l'environnement d'un noeud. Un attribut est défini par des fonctions associées à chaque production de la grammaire.

Les grammaires attribuées sont utilisées pour définir la sémantique statique en associant des attributs synthétisés à chaque symbole non terminal et des règles sémantiques à chaque production. L'attribution pourrait être faite en utilisant uniquement des attributs synthétisés. Toutefois, les attributs hérités fournissent des moyens de simplification. Ils permettent de déterminer le sens d'une construction en se basant sur le contexte dans lequel elle apparaît.

Les règles sémantiques sont *bien définies* si leur formulation permet toujours à tous les attributs d'être définis à tous les noeuds dans l'arborescence. Il est crucial que chaque règle d'une grammaire soit *bien définie*, car la quantité d'arborescences possibles est infinie. La notion de *bien défini* est énoncée formellement dans la référence [31]. Un algorithme permettant la vérification de ce critère est donnée dans la référence [12].

Pour la formalisation du langage ROOM, nous avons retenu le formalisme des grammaires attribuées. Toutefois, il en existe plusieurs autres. Les techniques telles l'algorithme markovien croissant de Bakker [1] et la sémantique dénotationnelle écrite en λ - calcul [13] [14] en sont des exemples. Cependant, le fait qu'elles soient exprimées par des processus définis sur des programmes exige la compréhension d'un compilateur du langage avant celle de la définition de ce langage. Les grammaires attribuées permettent

de définir les constructions d'un langage en faisant uniquement référence à leur contexte. Une telle technique conduit à une définition simple, structurée et concise.

Chapitre 4

Étude préliminaire à l'implantation de la sémantique statique

L'implantation de la grammaire développée dans le chapitre précédent se fait à partir d'analyseurs lexicaux et d'analyseurs syntaxiques. La combinaison de ces outils permet de vérifier si un texte fourni en entrée est conforme aux règles de grammaires du modèle ROOM. Lorsque une règle est reconnue, il est alors possible d'y associer le code correspondant à son implantation. Il existe plusieurs outils qui ont la capacité de produire ces analyseurs qui permettent la réalisation d'un compilateur. Ce chapitre présente une étude préliminaire de ces outils et les compare dans le but d'implanter notre grammaire attribuée. La première partie porte sur les différentes versions de la combinaison *lex* & *yacc* tandis que la deuxième partie aborde la boîte à outils Cocktail.

4.1 *lex* & *yacc*

Lex est un outil qui permet la génération d'analyseurs lexicaux. Ces derniers considèrent des données fournies en entrée et les divisent en jetons lexicaux selon une spécification donnée. Ces spécifications sont formées de règles qui se divisent en deux parties. La

partie de gauche contient des expressions régulières qui décrivent les jetons. La partie de droite contient les actions à entreprendre lors de la détection du jeton. *Lex* génère ses routines en C. Dans le programme principal, l'appel de la fonction *yylex()* permet d'obtenir le prochain jeton détecté. La fonction *yyinput()* permet d'obtenir les caractères qui suivent le dernier jeton obtenu. Lorsqu'on désire conserver des informations créées lors du traitement (comme les expressions non définies), la fonction *yyoutput()* permet d'écrire son argument dans un fichier de sortie. Et dans le cas où l'on désire retourner quelques caractères du jeton à la pile, la fonction *yyunput()* en rempile le nombre spécifié.

Pour l'analyse syntaxique, *yacc* permet de générer des routines en C qui accomplissent cette tâche. Moyennant une spécification grammaticale de type *lalr(1)* et des jetons fournis par un analyseur lexical, l'analyse syntaxique d'un extrait d'un langage peut être faite. Les symboles peuvent contenir des structures de données de tout type. Ces dernières sont appelées les attributs du symbole. Il est possible d'affecter des valeurs à ces attributs d'une manière synthétisée ou héritée. Ceci permet l'implantation d'une grammaire attribuée [31]. L'attribution synthétisée est réalisée lorsque les valeurs du symbole résultant (partie gauche de la règle) sont calculées à partir des valeurs des symboles de la partie droite de la règle. L'attribution héritée se fait lorsque les valeurs des symboles de la partie droite d'une règle sont issues de règles préalablement utilisées sur la pile. Le traitement automatique des erreurs syntaxiques dans *yacc* est plutôt limité. L'insertion de jeton *error* dans la grammaire permet d'établir des points de synchronisation. Lorsqu'une erreur survient, c'est à partir de ceux-ci que l'analyse est susceptible de continuer. Le succès de la reprise n'est pas assuré, car il est possible que l'état erroné ne soit pas complètement éliminé. Alors les erreurs continueront de s'empiler jusqu'à un état récupérable ou l'analyse s'arrêtera. La resynchronisation se fait comme suit :

1. L'erreur est signalée par la fonction *yyerror()*.

2. La reprise se fait en écartant toutes les règles partiellement traitées jusqu'à ce qu'il retourne dans un état où le jeton *error* peut être empilé.

3. L'analyse reprend en empilant le jeton *error*.

4. Les jetons suivants sont ignorés jusqu'au premier qui peut suivre le jeton *error* dans la grammaire.

5. Aucune erreur syntaxique subséquente n'est signalée jusqu'à ce qu'au moins trois symboles aient été empilés avec succès. Dans le cas où une erreur survient avant cette condition, l'exécution reprend à l'étape 2. Dans le cas contraire, l'état de l'analyseur est considéré comme resynchronisé et le traitement se poursuit normalement.

Toute fonctionnalité supplémentaire doit être programmée manuellement.

4.1.1 Les différentes formes de *lex*

Plusieurs versions de *lex* et de *yacc* sont disponibles sur le marché. Chacune comporte des particularités, avantages ou désavantages qui les différencient. Les tableaux 1 et 2 dressent des comparaisons techniques entre ces différentes versions.

Pour *lex*, la première implantation et la plus populaire est AT&T *lex*, ce qui par conséquent fait d'elle le standard de facto de l'outil. En conséquence, toute référence à *lex* dans ce texte porte sur cette version. Le projet GNU distribue sa propre version appelée *flex*. Naturellement, elle est disponible gratuitement. Contrairement à *lex*, *flex* utilise des tables internes dynamiques. Ceci évite d'avoir à augmenter manuellement la grosseur de la table des symboles lorsqu'on traite une spécification lexicale volumineuse. Selon [15] *flex* est beaucoup plus fiable que *lex*, qui contient plusieurs erreurs, et les analyseurs qu'il génère sont plus rapides. Ces deux versions ne sont disponibles que sous UNIX.

Pour les systèmes d'exploitation OS/2 et MS-DOS, deux implantations sont distribuées soient: *lex* de MKS et *pclex* de Abraxas. La première s'apparente à *lex* tandis que la deuxième s'apparente à *flex*. Les principales différences proviennent des particularités des systèmes d'exploitation utilisés.

4.1.2 Les différentes formes de *yacc*

Pour *yacc*, la première implantation et la plus populaire est également AT&T *yacc*. C'est aussi le standard de facto de l'outil et les références à *yacc* dans ce texte portent sur celle-ci. Similaire à *yacc*, Berkeley *yacc* est une extension de *yacc*. La contrepartie de *yacc* offerte par GNU s'appelle *bison* et est également disponible gratuitement. *Bison* est dérivé d'une ancienne version de Berkeley *yacc* mais, depuis 1988, est développé de façon indépendante. Il utilise de la mémoire dynamique alors que *yacc* utilise de la mémoire statique. Ceci évite d'avoir à augmenter manuellement la grosseur de la table des actions lorsqu'on traite une spécification syntaxique volumineuse. Plusieurs autres différences sont mentionnées dans la documentation de *bison*. Par exemple : la possibilité de créer un analyseur récursif pouvant tirer parti d'un environnement permettant le *multi-threading*, l'accès aux coordonnées de jetons (ligne et colonne) dans le fichier source, la possibilité de changer le préfixe *yy* des fonctions générées pour permettre l'utilisation de plus d'un analyseur syntaxique dans le même programme, etc. Les versions citées précédemment ne sont disponibles que sous UNIX.

Pour les systèmes d'exploitation OS/2 et MS-DOS, deux implantations sont distribuées soient: *yacc* de MKS et *pcyacc* de Abraxas. Les principales différences proviennent des particularités des systèmes d'exploitation utilisés.

Il existe même une norme IEEE POSIX P1003.2 définissant *lex* et *yacc*. Dans le cas de *lex*, elle ressemble beaucoup à *flex* avec quelques différences comme la possibilité de traiter des expressions régulières contenant des caractères accentués ou graphiques. Pour ce qui est de *yacc*, elle s'apparente à Berkeley *yacc* sauf pour les noms de fichiers de sortie qui sont ceux d'AT&T *yacc*.

4.2 *Cocktail*

Cocktail est un ensemble d'outils rassemblés pour la construction de compilateurs.

Les principaux éléments qui le composent sont :

- *Rex* - un générateur d'analyseurs lexicaux.
- *Lalr* et *Ell* - des générateurs d'analyseurs syntaxiques.
- *Ast* - un générateur d'arbres syntaxiques abstraits.
- *Ag* - un générateur d'évaluateurs de grammaires attribuées ordonnées.
- *Puma* - un outil de transformations d'arbres attribués basé sur la reconnaissance de patron et l'unification.
- BEG - un *back-end generator*

ainsi qu'une collection de préprocesseurs qui permettent des traductions de spécifications.

4.2.1 *Rex*

Rex est un générateur d'analyseurs lexicaux. Il utilise une spécification qui lui indique les jetons possibles ainsi que les actions à entreprendre lorsqu'ils sont détectés. La spécification détaille les composants lexicaux d'un langage à l'aide d'expressions régulières. *Rex* génère des routines (en C ou en Modula-2) qui permettent la segmentation lexicale de code source en entrée. La partie gauche des règles contient les états de départ et la partie droite les expressions régulières. Si plusieurs expressions régulières correspondent aux caractères en entrée, la correspondance la plus longue sera retenue. Cependant, s'il y a toujours plus qu'une correspondance possible, la première sera retenue. Selon Grosch [8], l'aspect le plus intéressant de *Rex* est sa capacité à générer des analyseurs rapides et petits.

Selon Grosch, *Rex* est 5 fois plus rapide et 5 fois plus petit que les analyseurs générés par *lex*. Il peut également créer automatiquement un programme exécutable pour tester l'analyseur. Selon lui, voici les avantages de *Rex* par rapport à *lex*:

- L'état initial a un nom documenté: STD (INITIAL pour *yacc*).
- Une liste d'états peut être inversée par l'opérateur NOT. Ceci indique que la règle est valable pour tous les états sauf ceux mentionnés.
- La spécification peut être écrite sans format prédéfini (les espaces sous forme d'espaces, tabulations et retours de chariot sont ignorés).
- Les identificateurs référant à des expressions régulières nommées ne sont pas compris entre accolades.
- *Rex* calcule automatiquement la position des jetons dans les champs *Line* et *Column* de la variable *<Scanner>_Attribute*.
- Il y a des règles prédéfinies pour ignorer les espaces.
- Les fichiers inclus (*include files*) peuvent être imbriqués jusqu'à un niveau de profondeur de 15 (un seul niveau dans *yacc*).
- Des routines sont fournies pour normaliser les jetons en minuscule ou majuscule.
- Aucun ajustement des structures de données internes n'est nécessaire pour le traitement de spécifications volumineuses.

Et les désavantages de *Rex* par rapport à *lex*:

- *yymore()* n'est pas disponible pour obtenir le prochain jeton.
- REJECT n'est pas disponible - *Rex* ne trouve qu'une seule solution et ne peut traiter le même segment de texte plus d'une fois.

- La redirection de l'entrée à partir de *yywrap()* n'est pas disponible. A la fin du fichier d'entrée, on ne peut que s'arrêter.

- Le jeu de caractères est limité à celui de l'ordinateur utilisé.

4.2.2 *Lalr*

Lalr permet de créer des analyseurs syntaxiques. Comme son nom l'indique, il traite les grammaires de types *lalr(1)*. Ces grammaires peuvent être écrites en EBNF. On peut associer aux règles de grammaire des actions décrites par des énoncés du langage cible (C ou Modula-2). Lorsqu'une règle est reconnue par l'analyseur, le code associé est exécuté. Un mécanisme pour l'attribution synthétisée (*S-attribution*) est fourni pour permettre la communication entre les actions. Plus exactement, les attributs de la partie gauche d'une règle ne peuvent être calculés qu'à partir des attributs de la partie droite de la règle. Les attributs de règles précédentes ne peuvent être utilisés.

Lors de conflits de réduction à gauche, un arbre de dérivation est imprimé pour faciliter la localisation du problème. Les erreurs syntaxiques sont traitées automatiquement par l'analyseur. Ce dernier génère des fonctions qui signalent les erreurs, assument la reprise ainsi que la correction lorsque possible. La méthode *backtrack-free* de [22] [23] est celle utilisée. Les analyseurs syntaxiques générés sont basés sur des tables (*table-driven*). Selon les auteurs, ils sont 2 à 3 fois plus rapides que ceux générés par *yacc*. Leur grosseur est cependant légèrement supérieure afin de permettre cette rapidité.

4.2.3 *Ell*

Ell est un outil qui permet la création d'un analyseur syntaxique qui traite des grammaires de type *ll(1)*. Les spécifications qu'il utilise peuvent être écrites en EBNF. Un mécanisme pour l'attribution héritée et synthétisée évaluée lors d'une seule traversée

en pré-ordre (*L-attribution*) est fourni. Les erreurs syntaxiques sont traitées automatiquement par l'analyseur qui les signale, assume la reprise et la correction (lorsque possible). Si la grammaire peut se restreindre au type *LL(1)*, on bénéficie alors d'un analyseur très rapide (900 000 lignes par minute).

4.2.4 *Ast*

Ast offre la possibilité de créer du code permettant de définir la structure d'arbres syntaxiques abstraits ainsi que des procédures en permettant la manipulation. Ces arbres peuvent être décorés d'attributs de types arbitraires. Les arbres ne sont pas les seuls structures permises, les graphes peuvent également être traités par *Ast*.

4.2.5 *Ag*

Ag offre la possibilité de créer du code permettant d'évaluer des attributs. Il permet de traiter des grammaires *bien définies* représentées sous forme d'arbres syntaxiques abstraits. Les terminaux et les non-terminaux peuvent avoir un nombre arbitraire d'attributs. *Ag* permet des attributs définis localement aux règles ou hérités de façon simple ou multiple. L'évaluation des attributs est exprimée dans le langage cible et dans un style fonctionnel.

4.2.6 Préprocesseurs

La collection de préprocesseurs offerts dans *Cocktail* se divise en deux familles. La première contient les préprocesseurs *cg* et *rpp* qui permettent respectivement la dérivation de spécifications pour un analyseur lexical et syntaxique à partir d'une grammaire attribuée. La deuxième offre des traducteurs de spécifications qui vont de *lex* vers *Rex* et de *yacc* vers *Lalr* ainsi que leurs contreparties. Tous les préprocesseurs agissent comme des filtres qui lisent de l'entrée standard et écrivent à la sortie standard.

4.3 Recommandations

Cette étude a permis l'identification des outils les plus appropriés, parmi ceux étudiés, pour l'implantation de la grammaire attribuée du langage ROOM. La grammaire obtenue au chapitre précédent pourrait facilement être mise en forme *ll(1)* ou *lalr(1)* et l'évaluation des outils tient compte de ce facteur. Les recommandations pour l'analyse lexicale et syntaxique sont maintenant présentées.

4.3.1 Analyse lexicale

Au niveau de l'analyse lexicale, *flex* est l'outil qui est le plus adapté aux besoins de l'implantation. *Rex* pourrait également être retenu, mais les caractéristiques qui les départagent sont assez importantes pour que *flex* soit favorisé. La limite de 15 niveaux que *Rex* impose sur l'imbrication de fichiers est suffisante pour la majorité des cas. Cependant, l'absence d'une telle restriction dans *flex* permet de traiter un éventail plus large de modèles. La fonction *yymore()* offerte par *flex* n'est pas utile pour les grammaires *ll(1)* ou *lalr(1)* et peut donc être ignorée. L'action *REJECT* que *Rex* ne supporte pas permet d'évaluer plusieurs règles pour un même segment, ce qui évite à l'analyseur d'arriver à une mauvaise conclusion trop rapidement. Lors d'erreurs, elle permet de continuer le traitement avec une autre règle qui pourrait aussi être satisfaite. La fonction *yywrap()* de *flex* permet de continuer la lecture des jetons dans un autre fichier lorsque la fin du ficher courant est atteinte. Ceci est utile pour compiler plusieurs modèles à la fois. Les quatre caractéristiques du tableau 1, que *Rex* possède mais que *flex* ne possède pas, ne sont pas essentielles ou peuvent être facilement reproduites.

4.3.2 Analyse syntaxique

Pour la création de l'analyseur syntaxique, l'outil *Ell* de *Cocktail* est recommandé. Cependant, il faut préciser que la grammaire doit être en forme *ll(1)*. Voici les principaux facteurs qui font de *Ell* le meilleur outil. L'utilisation des préprocesseurs de *Cocktail* permet de pouvoir traiter directement la forme linéaire produite par ObjecTime. La correction des erreurs syntaxiques de façon automatique produit un compilateur plus efficace. Le traitement de l'attribution héritée permet de tirer partie de cette facette qui est présente dans les grammaires attribuées. Les structures de données internes dynamiques ne limitent pas la taille du problème à traiter. La possibilité d'inclure plusieurs analyseurs syntaxiques pourrait être utile dans l'analyse des expressions en RPL ou C++. A ceci s'ajoute la possibilité d'utiliser d'autres outils de *Cocktail* tels *Ast*, *Ag*, *Puma*, et *Beg*.

	lex	*flex*	*Rex*
Création automatique d'un programme pilote de test			x
Opérateur NOT pour une liste d'états			x
Calcul automatique de la position des jetons			x
Niveau maximum d'imbrication de fichiers inclus	1	illimité	15
Routines fournies pour normalisation des jetons en minuscules ou majuscules			x
Structures de données internes dynamiques		x	x
yymore()	x	x	
Action REJECT	x	x	
yywrap()	x	x	
Jeu de caractères limité à celui de l'ordinateur utilisé		x	x
Support du code ISO 8 bits		x	x

TAB. 1 – *Sommaire comparatif des outils de création d'analyseurs lexicaux*

	yacc	*bison*	*Lalr*	*Ell*
Type de grammaire traitée	*lalr(1)*	*lalr(1)*	*lalr(1)*	*ll(1)*
Traitement des grammaires EBNF			préproc.	préproc.
Reprise automatique à la suite d'erreurs	x	x	x	x
Correction automatique d'erreurs syntaxiques			x	x
Attribution synthétisée	x	x	x	x
Attribution héritée	x	x		x
Structures de données internes dynamiques		x	x	x
Calcul automatique de la position des jetons		x	x	x
Analyseurs multiples à l'intérieur d'un même programme		x	x	x
Analyseur récursif		x		
Création automatique d'un programme exécutable pour tester			x	x

TAB. 2 – *Sommaire comparatif des outils de création d'analyseurs syntaxiques*

Conclusion

Les contributions originales apportées par la recherche présentée dans ce mémoire sont un logiciel orienté objet de poursuite en temps réel de satellites, une grammaire attribuée du langage ROOM ainsi qu'une étude sur les outils pouvant servir à son implantation. La création de SatSy a permis de documenter l'analyse et la conception d'un système temps-réel qui a été implanté et testé. Cette réalisation a également donné lieu à un article [17] publié dans le colloque ARRL/TAPR Digital Communications Conference (Seattle, septembre 1996). À cette occasion, notre contribution s'est méritée le prix du meilleur article à caractère technique et théorique par un étudiant. L'élaboration de la grammaire attribuée a, quant à elle, mis en évidence quelques incohérences de la syntaxe du langage ROOM. Par conséquent, nous avons contribué à l'amélioration de la qualité et de la fiabilité du langage ROOM. Cet exercice fournit également, en partie, une base formelle à une méthodologie de plus en plus utilisée. L'étude préliminaire présentée à la fin de ce mémoire permettra de faciliter l'implantation d'un compilateur du langage ROOM.

Le travail effectué dans ce mémoire présente certaines limites. Ainsi le logiciel SatSy est incapable de poursuivre des satellites en orbite elliptique et ne comporte pas une interface graphique avancée. Du côté de la grammaire attribuée, les expressions écrites dans le langage détaillé ne sont pas traitées.

Le logiciel SatSy a jeté les bases d'un nouveau projet de poursuite en temps réel de satellites. En effet, nous travaillons sur la mise en oeuvre d'un logiciel qui rendra

disponible sur le réseau Internet les ressources d'une station de télécommunications par satellites (http://www.dmi.usherb.ca/~barbeau/Radio). L'installation d'une telle station exige des compétences techniques et un investissement important. Grâce à notre logiciel et l'Internet, l'accès à une telle station sera ouvert à une large communauté. Cette application permettra essentiellement à des personnes branchées à l'Internet de dialoguer avec les occupants d'engins spatiaux, dont la navette spatiale américaine, la station orbitale russe Mir et la future station spatiale internationale. La communication sol-engin spatial s'effectue au moyen de fréquences radio dans les bandes radio amateur. Des équipements radio amateur sont installés dans les engins spatiaux afin de détendre les membres d'équipage. Les agences spatiales ont établi des programmes qui permettent à des étudiants de différents niveaux scolaires de dialoguer avec les astronautes durant les vols. La participation d'une école à une telle expérience exige l'installation sur place d'équipement de télécommunications relativement complexe. Le logiciel que nous développons, qui découle de notre expérience avec SatSy, va simplifier considérablement la tâche puisqu'il exige de la part des participants uniquement un ordinateur et un accès à l'Internet.

Annexe A

La grammaire attribuée du langage ROOM

A.1 Attribute Grammar of the ROOM Language

La grammaire présentée dans cette annexe a été produite en anglais dans le but d'être utilisée par la firme ObjecTime d'Ottawa.

```
type
    definition_class = (object_definition, type_definition, unknown_definition);
    definition = record
        uid : integer;
        ident : symbol;
        case k : definition_class of
            object_definition : (object_type : mode);
            type_definition : (defined_type : mode);
    end;
    definition_table = ↑ dt_element;
```

```
    dt_element = record
        first : definition ;
        rest : definition_table
    end ;

    value_table = ↑ dt_value ;
    dt_value = record
        first : symbol ;
        rest : value_table
    end ;

    modes = ↑ mode_value ;
    mode_value = record
        first : mode;
        rest : modes
    end ;

type_class = (actor_class_spec_type, port_ref_item_type, actor_ref_item_type,
    binding_item_type, equiv_item_type, sap_ref_item_type, function_spec_type,
    var_rpl_type, var_c++_type, state_item_type, choicepoint_item_type,
    transition_item_type, choicept_end_type, event_item_type, protocol_class_type,
    message_type_spec_type, data_spec_type, choice_type_spec_type, sequence_spec_type,
    seq_field_spec_type, seq_of_spec_type, real_spec_type, integer_spec_type,
    string_numeric_spec_type, string_printable_spec_type, constant_spec_type,
    const_type_spec_type, identified_type, unidentified_type) ;

mode = ↑ record
        case = k : type_class of
            actor_class_spec_type : ( actor-superclass-id : mode,
                genericity-spec : value_table,
```

actor-interface-spec : definition_table,

actor-implem-spec : definition_table) ;

port_ref_item_type : (replication-factor-S : value,

conjugated-S : value,

protocol-class-name-use : mode) ;

actor_ref_item_type : (replication-factor-S : value,

dynamics-type : value,

substitutable-S : value,

actor-class-name-use : mode) ;

binding_item_type : (replication-factor-S : value,

end-pt-spec : definition_table,

to-end-pt-spec-list : definition_table) ;

equiv_item_type : (paths-list : definition_table) ;

sap_ref_item_type : (replication-factor-S : value,

conjugated-S : value,

sap-class-name-use : mode) ;

function_spec_type : (function-attributes-S : value_table,

function-code : alphanumeric) ;

var_rpl_type : (var-rpl-class-desc : mode) ;

var_c++_type : (replication-factor-list : value_table,

data-class-name-use : mode,

initargs-S : definition_table) ;

state_item_type : (state-spec-S : definition_table) ;

choicepoint_item_type : (expression-S : definition_table) ;

transition_item_type : (trans-source-pt : definition_table,

trans-dest-pt : definition_table,

trans-triggers-S : definition_table,

```
        trans-code-S : definition_table) ;
    choicept_end_type : ( true-false-branch : value_table) ;
    protocol_class_type : ( protocol-superclass-id : mode,
        protocol-spec : definition_table) ;
    message_type_spec_type : ( data-class-name-use : mode) ;
    data_spec_type : ( data-superclass-id : mode,
        data-type-spec : definition_table,
        methods-spec-S : definition_table) ;
    choice_type_spec_type : ( type-names-list-S : mode) ;
    sequence_spec_type : ( seq-fieldlist-S : definitions) ;
    seq_field_spec_type : ( data-class-name-use : mode,
        optional-S : value,
        default-S : value) ;
    seq_of_spec_type : ( replication-factor : value,
        data-class-name-use : mode) ;
    integer_spec_type : ( minimum-S : value,
        maximum-S : value) ;
    real_spec_type : ( minimum-S : value,
        maximum-S : value) ;
    string_numeric_spec_type : ( replication-factor : value);
    string_printable_spec_type : ( replication-factor : value);
    constant_spec_type : ( isa-constant-spec-type-S : mode) ;
    const_type_spec_type : ( valid-const-type : value,
        constant-literal : value) ;
    identified_type : ( definition : integer) ;
    unidentified_type : ( identifier : symbol)
end ;
```

rule letter ::= *'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|*
'z'|'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'|'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|
'W'|'X'|'Y'|'Z'.

rule digit ::= *'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'*.

rule non-zero-digit ::= *'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'*.

rule natural-star ::= non-zero-digit.

rule natural-star ::= natural-star digit.

rule replication-factor ::= natural-star.

rule positive-integer ::= digit.

rule positive-integer ::= positive-integer digit.

rule integer ::= positive-integer.

rule integer ::= '-' positive-integer.

rule real ::= integer '.' positive-integer.

rule alphanumeric ::= letter.

rule alphanumeric ::= positive-integer.

rule ROOM-identifier ::= letter.

rule ROOM-identifier ::= ROOM-identifier alphanumeric.

rule class-name ::= ROOM-identifier.

rule root-id ::= 'root'.

rule sap-class-name ::= 'Frame'.

attribution

sap-class-name.mode ← N_mode(frame_type);

rule sap-class-name ::= 'Timing'.

attribution

sap-class-name.mode ← N_mode(timing_type);

rule sap-class-name ::= 'SimulationTiming'.

attribution

sap-class-name.mode ← N_mode(simulationTiming_type);

rule sap-class-name ::= 'Exception'.

attribution

sap-class-name.mode ← N_mode(exception_type);

rule sap-class-name ::= 'Log'.

attribution

sap-class-name.mode ← N_mode(log_type);

rule asn1-type-name ::= 'Boolean'.

attribution

asn1-type-name.mode ← N_mode(boolean_type);

rule asn1-type-name ::= 'Character'.

attribution

asn1-type-name.mode ← N_mode(character_type);

rule asn1-type-name ::= 'Integer'.

attribution

asn1-type-name.mode ← N_mode(integer_type);

rule asn1-type-name ::= 'Null'.

attribution

asn1-type-name.mode ← N_mode(null_type);

rule asn1-type-name ::= 'Real'.

attribution

asn1-type-name.mode ← N_mode(real_type);

rule asn1-type-name ::= 'SequenceOf'.

attribution

asn1-type-name.mode ← N_mode(sequenceOf_type);

rule asn1-type-name ::= 'String'.

attribution

asn1-type-name.mode ← N_mode(string_type);

rule asn1-type-name ::= 'Time'.

attribution

asn1-type-name.mode ← N_mode(time_type);

rule ROOM-type-name ::= 'RTActorId'.

attribution

ROOM-type-name.mode ← N_mode(rTActorId_type);

rule ROOM-type-name ::= 'RTByteBlock'.

attribution

ROOM-type-name.mode ← N_mode(rTByteBlock_type);

rule ROOM-type-name ::= 'RTMessage'.

attribution

ROOM-type-name.mode ← N_mode(rTMessage_type);

rule ROOM-type-name ::= 'RTPointer'.

attribution

ROOM-type-name.mode ← N_mode(rTPointer_type);

rule ROOM-type-name ::= 'RTSap'.

attribution

ROOM-type-name.mode ← N_mode(rTSap_type);

rule rpl-type-name ::= 'Array'.

attribution

rpl-type-name.mode ← N_mode(array_type);

rule rpl-type-name ::= 'Bag'.

attribution

rpl-type-name.mode ← N_mode(bag_type);

rule rpl-type-name ::= 'Dictionary'.

attribution

rpl-type-name.mode ← N_mode(dictionary_type);

rule rpl-type-name ::= 'IdentityDictionary'.

attribution

rpl-type-name.mode ← N_mode(identityDictionary_type);

rule rpl-type-name ::= 'IdentitySet'.

attribution

rpl-type-name.mode ← N_mode(identitySet_type);

rule rpl-type-name ::= 'Interval'.

attribution

rpl-type-name.mode ← N_mode(interval_type);

rule rpl-type-name ::= 'LinkedList'.

attribution

rpl-type-name.mode ← N_mode(linkedList_type);

rule rpl-type-name ::= 'OrderedCollection'.

attribution

rpl-type-name.mode ← N_mode(orderedCollection_type);

rule rpl-type-name ::= 'SortedCollection'.

attribution

rpl-type-name.mode ← N_mode(sortedCollection_type);

rule rpl-type-name ::= 'Set'.

attribution

rpl-type-name.mode ← N_mode(set_type);

rule rpl-type-name ::= 'Random'.

attribution

rpl-type-name.mode ← N_mode(random_type);

rule rpl-type-name ::= 'Bernoulli'.

attribution

rpl-type-name.mode ← N_mode(bernoulli_type);

rule rpl-type-name ::= 'Binomial'.

attribution

rpl-type-name.mode ← N_mode(binomial_type);

rule rpl-type-name ::= 'Exponential'.

attribution

rpl-type-name.mode ← N_mode(exponential_type);

rule rpl-type-name ::= 'Gamma'.

attribution

rpl-type-name.mode ← N_mode(gamma_type);

rule rpl-type-name ::= 'GeometricDistribution'.

attribution

rpl-type-name.mode ← N_mode(geometricDistribution_type);

rule rpl-type-name ::= 'Normal'.

attribution

rpl-type-name.mode ← N_mode(normal_type);

rule rpl-type-name ::= 'Poisson'.

attribution

rpl-type-name.mode ← N_mode(poisson_type);

rule rpl-type-name ::= 'SampleSpace'.

attribution

rpl-type-name.mode ← N_mode(sampleSpace_type);

rule rpl-type-name ::= 'Uniform'.

attribution

rpl-type-name.mode ← N_mode(uniform_type);

rule optionl-rpl-type-name ::= 'Association'.

attribution

optionl-rpl-type-name.mode ← N_mode(association_type);

rule optionl-rpl-type-name ::= 'Date'.

attribution

optionl-rpl-type-name.mode ← N_mode(date_type);

rule optionl-rpl-type-name ::= 'Fraction'.

attribution

optionl-rpl-type-name.mode ← N_mode(fraction_type);

rule optionl-rpl-type-name ::= 'Link'.

attribution

optionl-rpl-type-name.mode ← N_mode(link_type);

rule optionl-rpl-type-name ::= 'Point'.

attribution

optionl-rpl-type-name.mode ← N_mode(point_type);

rule valid-const-type ::= 'Boolean'.

attribution

valid-constant-type.mode ← N_mode(boolean_type);

rule valid-const-type ::= 'Character'.

attribution

valid-constant-type.mode ← N_mode(character_type);

rule valid-const-type ::= 'Integer'.

attribution

valid-constant-type.mode ← N_mode(integer_type);

rule valid-const-type ::= 'Real'.

attribution

valid-constant-type.mode ← N_mode(real_type);

rule valid-const-type ::= 'String'.

attribution

valid-constant-type.mode ← N_mode(string_type);

rule service-name ::= 'BasicCommunicationService'.

attribution

service-name.value ← BasicCommunicationService;

rule

service-name ::= 'SimulationCommunicationService'.

attribution

service-name.value ← SimulationCommunicationService;

The service name is used to specify if the normal communications (*BasicCommunicationService*) should take place or if latency in communications (*SimulationCommunicationService*) is desired in the communication environment.

A.1.1 Actor Classes

rule model ::= class-specification-list actor-class-name-use.

attribution

class-specification-list.environment ←

complete_env(class-specification-list.definitions)

condition

included(actor-class-name-use.symbol, class-specification-list.environment) **and**

unambiguous(class-specification-list.definitions) **and**

completed(class-specification-list.environment);

A *model* is composed of a list of class specifications *class-specification-list* amongst which one of them is the container class. This container class is of type actor and is identified by the symbol *actor-class-name-use*. The verification of the presence of this class in the *class-specification-list*, is done by the *included* predicate. Every definition of the *class-specification-list* must be included only once in the environment. The *complete_env* function searches the definition list for *unidentified* types and *unknown* definitions and tries to resolve these references. The function *unambiguous* insures that this condition is true. The function *completed* verifies that the environment is completely defined by checking for the absence of undefined types.

rule class-specification-list ::= class-specification.

rule class-specification-list ::= class-specification-list ';' class-specification.

attribution

class-specification-list[1].definitions ← class-specification-list[2].definitions &
class-specification.definitions ;

A list of class specifications is composed of one or several class specifications.

rule actor-class-name-use ::= actor-class-name.

attribution

actor-class-name-use.mode ← N_mode(unidentified_type,
actor-class-name.symbol);

rule class-specification ::= actor-class-spec.

rule class-specification ::= protocol-class-spec.

rule class-specification ::= data-class-spec.

A *class-specification* is either an actor class which defines active entities within the system, a protocol class which defines how actors communicate or a data class which defines the data containers of the system.

rule actor-class-spec ::= 'actor' 'class' actor-class-name

'derived' 'from' actor-superclass-id genericity-spec

'{' actor-interface-spec actor-implem-spec '}' ';'.

attribution

actor-class-spec.definitions ← N_definition(gennum,

actor-class-name.symbol,

type_definition,

N_mode(actor_class_spec_type,

actor-superclass-id.mode,

genericity-spec.values,

actor-interface-spec.definitions,

actor-implem-spec.definitions));

actor-implem-spec.environment ← complete_env(actor-interface-spec.definitions)

& complete_env(actor-implem-spec.definitions)

& actor-class-spec.environment;

condition

unambiguous(actor-interface-spec.definitions) **and**

actor-class-name.symbol ≠ actor-superclass-id.symbol;

An actor class specifies an abstraction which plays an active role in the model. The value of *actor-class-name* uniquely identifies the class. *Actor-superclass-id* represents a class from which it can inherit properties. Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship. In ROOM all attributes, operations, and behavior are inherited. Any inherited property can be redefined. The *genericity-spec* allows to specify a substitution rule for the actor. An actor communicates with other actors through ports. The list of ports that are at the boundary of an actor class specifies its interface which is defined by *actor-interface-spec.* The structure and behavior of an actor class is defined by *actor-implem-spec.*

rule actor-class-name ::= class-name.

rule actor-superclass-id ::= actor-class-name-use.

rule actor-superclass-id ::= root-id.

attribution actor-superclass-id.mode ← **Nil**;

An *actor-superclass-id* represents the identity of a superclass in an inheritance tree. It can take the value of *actor-class-name* for a defined class or *root-id* when no inheritance is applied.

rule genericity-spec ::= genericity-spec-is-sub genericity-spec-can-sub.

attribution

genericity-spec.values ← genericity-spec-is-sub.value &
genericity-spec-can-sub.value ;

rule genericity-spec-is-sub ::= 'is substitutable'.

attribution

genericity-spec-is-sub.value ← is_substitutable;

rule genericity-spec-is-sub ::= .

attribution

genericity-spec-is-sub.value ← **Nil**;

rule genericity-spec-can-sub ::= 'can substitute'.

attribution

genericity-spec-can-sub.value ← can_substitute;

rule genericity-spec-can-sub ::= .

attribution

genericity-spec-can-sub.value ← **Nil**;

A genericity specification of value *is substitutable* allows the class to be replaced by another one of compatible interface definition. The value *can substitute* means that this class definition can replace another one of compatible interface definition. A substituting actor has a compatible interface when it has at least the same ports as the substituted

actor. When the genericity specification is absent, no substitution is permitted in any way for that actor class.

rule actor-interface-spec ::= 'interface' ':' '{' port-ref-list-S '}' ';'.

An actor's interface specifies the means of communication of a class with its environment. It is made of ports that are placed at the boundary of the actor. These ports, which are references to protocols, enable communication with other actors. Two types of ports can be used. The first one is between the actor itself and other actors. The ports used in this situation are *end-ports*. The second one is between its component actors and other actors. The ports used in this case are *relay-ports*. The symbol *port-ref-list-S* indicates the possibility of a list of port definitions through which communication occur.

rule port-ref-list-S ::= .

rule port-ref-list-S ::= port-ref-list.

rule port-ref-list ::= port-ref-item ';'.

rule port-ref-list ::= port-ref-list port-ref-item ';'.

attribution

port-ref-list[1].definitions ← port-ref-list[2].definitions &
port-ref-item.definitions ;

A list of port references is defined by at least one or several port reference items symbolized by *port-ref-item*.

rule port-ref-item ::= 'port' port-ref-name replication-factor-S
'isa' conjugated-S protocol-class-name-use.

attribution

port-ref-item.definitions ← N_definition(gennum,
port-ref-name.symbol,
object_definition,
N_mode(port_ref_item_type,

replication-factor-S.value,

conjugated-S.value

protocol-class-name-use.mode));

A *port-ref-item* is a definition of a port. A port allows the flow of messages between actors. It is identified by the value of a *port-ref-name*. In the case where an actor has to communicate with several actors with the same messages, the port itself can be replicated by a *replication-factor-S* equal to the number of actors to reach. The protocol it uses to communicate is defined by the value of *protocol-class-name-use* and can be conjugated.

rule port-ref-name ::= ROOM-identifier ':'.

rule replication-factor-S ::= .

attribution

replication-factor-S.value ← **Nil**;

rule replication-factor-S ::= replication-factor.

attribution

replication-factor-S.value ← replication-factor.symbol;

A *replication-factor-S* indicates the possibility of replication. Replication enables-multiplicity of an entity without explicitly having to declare every incarnation.

rule conjugated-S ::= .

attribution

conjugated-S.value ← **Nil**;

rule conjugated-S ::= 'conjugated'.

attribution

conjugated-S.value ← conjugated;

A *conjugated-S* indicates the possibility of the conjugation of a port. When the value holds *conjugated*, it indicates that the direction in which the messages flow in a protocol is reversed. Thus enabling a conjugated port to receive the messages sent by another port

of the same protocol and vice versa.

rule protocol-class-name-use ::= protocol-class-name.

attribution

protocol-class-name-use.mode ← N_mode(unidentified_type,

protocol-class-name-use.symbol);

The use of a protocol class name implies that the refered class has previously been defined.

rule actor-implem-spec ::= 'implementation' ':' '{'structure-spec behavior-spec '}'.

attribution

actor-implem-spec.definitions ← structure-spec.definitions

& behavior-spec.definitions;

A class implementation specification defines the structure and behavior of an actor class.

Actor Classes - Structure

rule structure-spec ::= 'structure' ':' '{'

end-ports-spec-S

components-spec-S

bindings-spec-S

equivs-spec-S '}'.

attribution

structure-spec.definitions ← end-ports-spec-S.definitions

& components-spec-S.definitions & bindings-spec-S.definitions

& equivs-spec-S.definitions;

condition

unambiguous(end-ports-spec-S.definitions) **and**

unambiguous(components-spec-S.definitions) **and**

unambiguous(bindings-spec-S.definitions) **and**

unambiguous(equivs-spec-S.definitions);

The structure of an actor class can hold the following entities: end-ports, components, bindings, and equivalences. The end-ports that are part of the structure are used to communicate with component actors. The bindings are the links that associate two ports thus enabling communication between them. Equivalences are the set of paths that can be taken to access a component in the case where it is contained by more than one actor.

rule end-ports-spec-S ::= .

rule end-ports-spec-S ::= end-ports-spec.

It indicates the possibility of end port specifications.

rule components-spec-S ::= .

rule components-spec-S ::= components-spec.

It indicates the possibility of component specifications.

rule bindings-spec-S ::= .

rule bindings-spec-S ::= bindings-spec.

It indicates the possibility of binding specifications.

rule equivs-spec-S ::= .

rule equivs-spec-S ::= equivs-spec.

It indicates the possibility of equivalence specifications.

rule end-ports-spec ::= 'end' 'ports' ':' '{' port-ref-list-S '}'.

An *end-ports-spec* is defined as a port definition list. End-ports convey messages from and to component actors of the actor in which they are defined. They represent termination points on a communication channel.

rule components-spec ::= 'components' ':' '{' components-list '}'.

A *components-spec* is defined as a component list.

rule components-list ::= actor-ref-item ';'.

rule components-list ::= components-list actor-ref-item ';'.

attribution

components-list[1].definitions ← components-list[2].definitions &

actor-ref-item.definitions;

A *components-list* is defined as one or several actor references. It holds the references of actors that make a composed actor.

rule actor-ref-item ::= 'actor' actor-ref-name replication-factor-S

'isa' dynamics-type substitutable-S actor-class-name-use.

attribution

actor-ref-item.definitions ← N_definition(gennum,

actor-ref-name.symbol,

object_definition,

N_mode(actor_ref_item_type,

replication-factor-S.value),

dynamics-type.value,

substitutable-S.value

actor-class-name-use.mode));

condition

substitutable-S.value = substitutable ⇒

retrieve_definition(actor-class-name-use.symbol,

actor-ref-item.environment).defined_type.genericity-spec-is-sub.value

= is_substitutable;

An *actor-ref-item* is a reference to an actor class. It is identified by the value of *actor-ref-name*. An actor reference can be replicated. The class of the actor to be incarnated is

refered by *actor-class-name*. The manner in which the actor is created is specified by its *dynamics-type*. Substitution can be allowed with an incarnation of another class. Provided naturally, that this other class has a compatible interface. The *retrieve_definition* function finds the definition of an actor class name with its symbol and return it.

rule actor-ref-name ::= ROOM-identifier.

rule dynamics-type ::= 'fixed'.

attribution

dynamics-type.value ← fixed;

rule dynamics-type ::= 'optional'.

attribution

dynamics-type.value ← optional;

rule dynamics-type ::= 'imported'.

attribution

dynamics-type.value ← imported;

A *dynamics-type* indicates the manner in which an actor is to be incarnated. When it is fixed, the actor is created at the same time as the actor that contains its reference. When it is optional, the actor can be dynamically created by the behavior of its containing actor. In the case where it is imported, the reference of the actor represents a placeholder in which an existing actor can be dynamically inserted.

rule substitutable-S ::= .

attribution

substitutable-S.value ← **Nil**;

rule substitutable-S ::= 'substitutable'.

attribution

substitutable-S.value ← substitutable;

A *substitutable-S* indicates the possibility of substitution.

rule bindings-spec ::= 'bindings' ':' '{' bindings-list '}'.

A *bindings-spec* is defined by a binding list.

rule bindings-list ::= binding-item ';'.

rule bindings-list ::= bindings-list binding-item ';'.

attribution

bindings-list[1].definitions ← bindings-list[2].definitions &
binding-item.definitions;

A *bindings-list* is defined by one or several bindings.

rule binding-item ::= 'binding' binding-name replication-factor-S ':'
'{' end-pt-spec to-end-pt-spec-list '}'.

attribution

binding-item.definitions ← N_definition(gennum,
binding-name.symbol,
object_definition,
N_mode(binding_item_type,
replication-factor-S.value,
end-pt-spec.definitions,
to-end-pt-spec-list.definitions));

condition

unambiguous(end-pt-spec.definitions, to-end-pt-spec-list.definitions);

A binding represents the linkage of ports in order to build a communication channel between two actors. This channel can be made of several ports. The first and last ports are called end-ports because they terminate the channel. The ports in between are called relay-ports because they only convey the information they receive whithout altering the message. The relay-ports are necessary to cross levels of hierarchy greater than one between actors. An actor can communicate directly (without relay ports) with another

actor if it is at the same level of a hierarchy within an actor or if it is a sub-actor.

rule to-end-pt-spec-list ::= 'to' end-pt-spec.

rule to-end-pt-spec-list ::= to-end-pt-spec-list 'to' end-pt-spec.

attribution

to-end-pt-spec-list[1].definitions ← to-end-pt-spec-list[2].definitions &
end-pt-spec.definitions;

A *to-end-pt-spec-list* is defined by one or more end point specifications.

rule end-pt-spec ::= port-ref-name '/' actor-ref-name.

attribution

end-pt-spec.definitions ← N_definition(gennum,
port-ref-name.symbol & actor-ref-name.symbol,
object_definition,
Nil);

condition

lookup(port-ref-name.symbol, end-pt-spec.environment) **and**
lookup(actor-ref-name.symbol, end-pt-spec.environment)

rule end-pt-spec ::= port-ref-name '/' actor-class-name.

attribution

end-pt-spec.definitions ← N_definition(gennum,
port-ref-name.symbol & actor-class-name.symbol,
object_definition,
Nil);

condition

lookup(port-ref-name.symbol, end-pt-spec.environment) **and**
lookup(actor-class-name.symbol, end-pt-spec.environment)

An end point specification is defined by a port reference name and an actor reference name or a port reference name and an actor class name when the targeted actor is replicated. The *lookup* function ensures that the symbol refers to a valid definition. An ambiguous situation could occur if two actor references within the same actor were replicated and had the same port name.

rule equivs-spec ::= 'equivalences' ':' '{' equivs-list '}'.

An *equivs-spec* is defined by a list of equivalences.

rule equivs-list ::= equiv-item ';'.

rule equivs-list ::= equivs-list equiv-item ';'.

attribution

equivs-list[1].definitions ← equivs-list[2].definitions & equiv-item.definitions ;

An *equivs-list* is defined by one or more equivalences.

rule equiv-item ::= 'set' equiv-name':' '{' paths-list '}'.

attribution

```
equiv-item.definitions ← N_definition( gennum,
    equiv-name.symbol,
    object_definition,
    N_mode( equiv_item_type,
        paths-list.definitions) ;
```

An equivalence defines a set of paths through which a component actor can be found in case of multiple containment. These paths start with a component actor of the actor being defined. The next item in the path will be a component actor of the previous component actor. This path goes down the hierarchy until it reaches the actor that is contained in several actors.

rule equiv-name ::= ROOM-identifier.

rule paths-list ::= actor-path-id and-actor-path-id-list.

attribution

paths-list.definitions ← and-actor-path-id-list.definitions &

actor-path-id.definitions;

A paths list is defined by one *actor-path-id* and an *and-actor-path-id-list.*

rule and-actor-path-id-list ::= 'and' actor-path-id.

rule and-actor-path-id-list ::= and-actor-path-id-list 'and' actor-path-id.

attribution

and-actor-path-id-list[1].definitions ←

and-actor-path-id-list[2].definitions &

actor-path-id.definitions;

An *and-actor-path-id-list* is defined by one or more *actor-path-id.*

rule actor-path-id ::= actor-ref-name-use actor-ref-name-use-S.

attribution

actor-path-id.definitions ← actor-ref-name-use.definitions &

actor-ref-name-use-S.definitions;

An *actor-path-id* is defined by one or more actor reference names.

rule actor-ref-name-use-S ::= .

attribution

actor-ref-name-use-S.definitions ← **Nil**;

rule actor-ref-name-use-S ::= actor-ref-name-use-S '/' actor-ref-name-use.

attribution

actor-ref-name-use-S[1].definitions ← actor-ref-name-use-S[2].definitions &

actor-ref-name-use.definitions;

condition

acyclic(actor-ref-name-S.definitions);

An *actor-ref-name-use-S* indicates the possibility of a list of one or more actor reference names.

rule actor-ref-name-use ::= actor-ref-name.

attribution

actor-ref-name-use.definitions ← N_definition(gennum,
actor-ref-name.symbol,
unknown_definition);

Actor Classes - Behavior

rule behavior-spec ::= 'behavior' ':' '{'
language-spec
saps-spec-S
inclusions-spec-S
functions-spec-S
fsm-spec '}'.

attribution

behavior-spec.definitions ← language-spec.value & saps-spec-S.definitions
& inclusions-spec-S.definitions & functions-spec-S.definitions &
fsm-spec.definitions;
functions-spec-S.language ← language-spec.value;

condition

unambiguous(saps-spec-S.definitions) **and**
unambiguous(function-spec-S.definitions) **and**
inclusions-spec-S.modes $\neq \varepsilon \Longrightarrow$ language-spec.value = C++;

A *behavior-spec* gives the behavior of an actor as time passes and events occur. The

language-spec defines the detail level language. The *saps-spec-S* lists the services accessible to the actor. These services can be provided by the system or through a protocol reference. An *inclusions-spec-S* gives the possibility to include external C++ definitions. The *function-spec-S* is the list of functions written in detail level language which scope is the behavior specification. The explicit behavior of the actor is modeled by a hierarchical finite state machine specified by the *fsm-spec*. A condition verifies that if inclusions are used, the language specification's value is C++.

rule language-spec ::= 'language' ':' language-identifier.

A language specification is defined as a *language-identifier*.

rule language-identifier ::= 'RPL'.

attribution

language-identifier.value ← RPL;

rule language-identifier ::= 'C++'.

attribution

language-identifier.value ← C++ ;

rule language-identifier ::= 'Unspecified'.

attribution

language-identifier.value ← Unspecified ;

A language identifier can take on three values. The first two (RPL, C++) identify implementation languages allowed for the detail level language. The third value (Unspecified) indicates that the target language has not yet been chosen or is irrelevant.

rule saps-spec ::= 'saps' ':' '{' saps-list '}'.

A *saps-spec* is defined by a service access point list.

rule saps-list ::= sap-ref-item ';'.

rule saps-list ::= saps-list sap-ref-item ';'.

attribution

saps-list[1].definitions ← saps-list[2].definitions & sap-ref-item.definitions;

A *saps-list* is defined by one or several service access point references.

rule sap-ref-item ::= 'sap' sap-ref-name replication-factor-S
'isa' conjugated-S sap-class-name-use.

attribution

sap-ref-item.definitions ← N_definition(gennum,
sap-ref-name.symbol,
object_definition,
N_mode(sap_ref_item_type,
replication-factor-S.value,
conjugated-S.value,
sap-class-name-use.mode));

A *sap-ref-item* is a reference to a service access point. A sap represents a service an actor requires for it's implementation. It is identified by the value of a *sap-ref-name*. The way by which it communicates is determined by a *sap-class-name*. A service access point can be replicated and conjugated.

rule sap-ref-name::= ROOM-identifier.

rule sap-class-name-use ::= protocol-class-name-use.

A *sap-class-name* can be defined in six different ways. It can hold the value *Frame* which indicates that the sap communicates with the frame service. This service is used to dynamically add and remove actors in the model. When it holds the value *Timing*, it uses a service based on the expiration of the real-time clock. Whereas when it holds the value *SimulationTiming*, it uses a service based on a simulated abstract time which works in conjonction with the simulation environment. When the value equals *Exception*, an exception handling service is used to trap execution errors. It can also hold the value *Log* which indicates that the service invoked records system and application events. And

finally, it can be defined as a *protocol-class-name* in which case the sap is like an ordinary port. The services offered by the system are defined in the model's environment.

rule inclusions-spec-S ::= .

rule inclusions-spec-S ::= inclusions-spec.

An *inclusion-spec-S* indicates the possibility of an inclusion specification.

rule inclusions-spec ::= 'inclusions' ':' '{' inclusions-list '}'

An *inclusions-spec* is defined by an inclusion list.

rule inclusions-list ::= inclusion-spec ';'.

rule inclusions-list ::= inclusions-list inclusion-spec ';'.

attribution

inclusions-list[1].definitions ← inclusions-list[2].definitions &

inclusion-spec.definitions ;

An *inclusions-list* is defined by one or several inclusion specifications.

rule inclusion-spec ::= 'include' inclusion-name.

attribution

inclusions-spec.definitions ← ... ;

An inclusion specification is defined by an *inclusion-name*. The elements defined in the file *inclusion-name.symbol* are extracted from it and represented in structures of type *definition*. This portion of the grammar which is tied to the detail level language is not considered in this work.

rule inclusion-name ::= ROOM-identifier.

rule functions-spec-S ::= .

rule functions-spec-S ::= functions-spec.

An *function-spec-S* indicates the possibility of a function specification.

rule functions-spec ::= 'functions' ':' '{' functions-list '}'.

A function specification is defined by a function list.

rule functions-list ::= function-spec ';'.

rule functions-list ::= functions-list function-spec ';'.

attribution

functions-list[1].definitions ← functions-list[2].definitions &
function-spec.definitions;

A *functions-list* is defined by one or several function specifications.

rule function-spec ::= 'function' function-name':' function-attributes-S
'{' function-code '}'.

attribution

function-spec.definitions ← N_definition(gennum,
function-name.symbol,
object_definition,
N_mode(function_spec_type,
function-attributes-S.values,
function-code.text));

condition

function-attributes-S.definitions $\neq \varepsilon \Rightarrow$ function-spec.language = C++;

A function specification defines a function coded using the detail level language. Since functions are implemented in a different language, the verification of the code is beyond the scope of this work. The *function-attributes-S* indicates the possibility of specifiying the function's properties. They are only valid in the *C++* language. A condition verifies that this is true by checking the value of the *function-spec.language* attribute inherited from the *behavior-spec* rule.

rule function-name ::= ROOM-identifier-list ROOM-identifier.

attribution

function-name.symbol ← ROOM-identifier-list.symbol

& ROOM-identifier.symbol;

A *function-name* is defined as at least one *ROOM-identifier* and a *ROOM-identifier-list.*

rule ROOM-identifier-list ::= .

rule ROOM-identifier-list ::= ROOM-identifier-list ROOM-identifier':'.

attribution

ROOM-identifier-list[1].symbol ← ROOM-identifier-list[2].symbol &

ROOM-identifier.symbol;

A *ROOM-identifier-list* can be empty or hold one or several *ROOM-identifiers.*

rule function-attributes-S ::= .

rule function-attributes-S ::= function-attributes.

A *function-attributes-S* indicates the possibility of *function-attributes.*

rule function-attributes ::= 'attributes' ':' func-attrib-list.

condition

(public ∈ func-attrib-list.values ⊕ inheritable ∈ func-attrib-list.values ⊕

private ∈ func-attrib-list.values) **and**

(inline ∈ func-attrib-list.values ⊕ polymorphic ∈ func-attrib-list.values);

Function attributes are defined as a function attribute list. Among these, there are some incompatibilities. The attributes *public, inheritable,* and *private* are mutually exclusive. So are the attributes *inline* and *polymorphic.* The nature of these attributes is explained further on.

rule func-attrib-list ::= func-attrib.

rule func-attrib-list ::= func-attrib-list func-attrib.

attribution

func-attrib-list[1].values ← func-attrib-list[2].values & func-attrib.value;

A function attribute list is defined as one or several function attributes.

rule func-attrib ::= 'public'.

attribution

func-attrib.value ← public;

rule func-attrib ::= 'inheritable'.

attribution

func-attrib.value ← inheritable;

rule func-attrib ::= 'private'.

attribution

func-attrib.value ← private;

rule func-attrib ::= 'inline'.

attribution

func-attrib.value ← inline;

rule func-attrib ::= 'readOnly'.

attribution

func-attrib.value ← readOnly;

rule func-attrib ::= 'polymorphic'.

attribution

func-attrib.value ← polymorphic;

The *public* attribute indicates that the function can be used by any function. A *public* function allows to bypass ROOM's messaging system. When the value is *inheritable*, it can be used only by member functions of the same class and by member functions of derived classes. When the value is private, the function can only be accessed by member functions of the same class. The value *inline* is an indication to the compiler that the calls to this function should be replaced by the code that implements it. When the attribute's value is *readOnly*, the function has no side effect (for further details see Ref. [18]). And

finally, when the attribute's value is *polymorphic* the body of the function is to be supplied by a derived class.

rule fsm-spec ::= 'top' 'fsm' ':' '{' state-spec '}'.

A finite state machine specification is defined by a *state-spec*. Hierarchical finite state machines are used to specify the actor behavior when stimulated by external events or as time passes. The state specification defines a top level state that can be divided into substates.

rule state-spec ::= vars-spec-S

entry-action-spec-S

exit-action-spec-S

substates-spec-S

choicepoints-spec-S

transitions-spec-S.

attribution

state-spec.definitions ← vars-spec-S.definitions & entry-action-spec-S.definitions

& exit-action-spec-S.definitions & substates-spec-S.definitions

& choicepoints-spec-S.definitions & transitions-spec-S.definitions));

condition

unambiguous(vars-spec-S.definitions) **and**

unambiguous(substates-spec-S.definitions) **and**

unambiguous(choicepoints-spec-S.definitions) **and**

unambiguous(transitions-spec-S.definitions);

A state specification is defined by state variables, entry actions which are executed before entering the state, exit actions which are executed when the state is exited, substates which decompose a superstate into lower-level states, choicepoints which allow two paths out of a state upon the same event, and finally transitions which direct the

control flow towards other states.

rule vars-spec-S ::= .

rule vars-spec-S ::= vars-spec.

A *vars-spec-S* indicates the possibility of a variable specification.

rule entry-action-spec-S ::= .

rule entry-action-spec-S ::= entry-action-spec.

An *entry-action-spec-S* indicates the possibility of an entry action specification.

rule exit-action-spec-S ::= .

rule exit-action-spec-S ::= exit-action-spec.

An *exit-action-spec-S* indicates the possibility of an exit action specification.

rule substates-spec-S ::= .

rule substates-spec-S ::= substates-spec.

A *substates-spec-S* indicates the possibility of a substate specification.

rule choicepoints-spec-S ::= .

rule choicepoints-spec-S ::= choicepoints-spec.

A *choicept-spec-S* indicates the possibility of a choicepoint specification.

rule transitions-spec-S ::= .

rule transitions-spec-S ::= transitions-spec.

An *transitions-spec-S* indicates the possibility of a transition specification.

rule vars-spec ::= 'vars' ':' '{' vars-list '}'.

A *vars-spec* is defined by a variable list.

rule vars-list ::= var-item ';'.

rule vars-list ::= vars-list var-item ';'.

attribution

vars-list[1].definitions ← vars-list[2].definitions & var-item.definitions.

A *vars-list* is defined by one or several variables.

rule var-item ::= var-rpl.

rule var-item ::= var-c++.

A variable is defined either as an RPL variable or as a C++ variable.

rule var-rpl ::= var-name 'isa' var-rpl-class-desc.

attribution

var-rpl.definitions ← N_definition(gennum,

var-name.symbol,

object_definition,

var-rpl-class-desc.mode);

An RPL variable is defined by a variable name which identifies it uniquely and a variable class descriptor which specifies its type.

rule var-name ::= ROOM-identifier.

An RPL variable name is defines as a ROOM-identifier.

rule var-rpl-class-desc ::= data-type-name.

An RPL variable class name is defined as a *data-type-name.*

rule data-type-name ::= asn1-type-name.

rule data-type-name ::= ROOM-type-name.

rule data-type-name ::= rpl-type-name.

rule data-type-name ::= optionl-rpl-type-name.

rule data-type-name ::= user-defined-rpl-type-name.

rule user-defined-rpl-type-name ::= data-class-name-use.

rule data-class-name-use ::= data-class-name.

attribution

data-class-name-use.mode ← N_mode(unidentified_type,

data-class-name.symbol);

rule var-c++ ::= var-name replication-factor-list

'isa' data-class-name-use initargs-S.

attribution

var-c++.definition ← N_definition(gennum,

var-name.symbol,

object_definition,

N_mode(var_c++_type,

replication-factor-list.values,

data-class-name-use.mode,

initargs-S.definitions));

A c++ variable is defined by a *var-name* which identifies it uniquely, a c++ class name which specifies its type, a replication factor list which gives the possibility to build multidimensional arrays of the class type, and arguments which allows initialization of the variables with predefined values. The assignment of the mode attribute of the c++-class-name-use symbol is part of the attribute grammar of the detail level language.

rule replication-factor-list ::= .

rule replication-factor-list ::= replication-factor replication-factor-list.

attribution

replication-factor-list[1].values ← replication-factor-list[2].values &

replication-factor.symbol;

A replication factor list allows the creation of multidimensional array variables.

rule initargs-S ::= .

rule initargs-S ::= 'initargs' '{' c++-argument-list '}'.

An *initargs-S* indicates the possibility of an argument list for the initialization of the variables. The evaluation of the c++-argument-list symbol is part of the attribute

grammar of the detail level language.

rule entry-action-spec ::= 'entry' 'action' ':' '{' expression-S '}'.

An entry action specification is defined by the possibility of an expression. This expression, when present, is evaluated before entering the state in which it is defined.

rule exit-action-spec ::= 'exit' 'action' ':' '{' expression-S '}'.

An exit action specification is defined by the possibility of an expression. This expression, when present, is evaluated while leaving the state in which it is defined.

rule expression-S ::= .

attribution

expression.definitions ← **Nil**;

rule expression-S ::= expression.

attribution

expression-S.definitions ← ...;

An *expression-S* indicates the possibility of an expression. An expression is coded in the detail level language.

rule expression ::= rpl-expression.

rule expression ::= c++-expression.

An expression can be defined as an RPL expression or as a C++ expression. The evaluation of C++ or RPL expressions is part of the attribute grammar of the detail level language.

rule substates-spec ::= 'substates' ':' '{' states-list '}'.

A *substates-spec* is defined by a state list. These states make a finite state machine of a lower level within the super-state.

rule states-list ::= state-item ';'.

rule states-list ::= states-list state-item ';'.

attribution

states-list[1].definitions ← states-list[2].definitions & state-item.definitions.

A *states-list* is defined by one or several states.

rule state-item ::= 'state' state-name':' '{' state-spec-S '}'.

attribution

state-item.definitions ← N_definition(gennum,
state-name.symbol,
object_definition,
N_mode(state_item_type,
state-spec-S.definitions)) ;

A state is defined by the possibility of a state specification. It is also uniquely identified by a state name.

rule state-spec-S ::= .

rule state-spec-S ::= state-spec.

A *state-spec-S* indicates the possibility of a state specification.

rule state-name ::= ROOM-identifier;

rule choicepoints-spec ::= 'choicepoints' ':' '{' choicepoint-list '}'.

A *choicepoints-spec* is defined by a choicepoint list.

rule choicepoint-list ::= choicepoint-item ';'.

rule choicepoint-list ::= choicepoint-list choicepoint-item ';'.

attribution

choicepoint-list[1].definitions ← choicepoint-list[2].definitions &
choicepoint-item.definitions.

A *choicepoint-list* is defined by one or several choicepoints.

rule choicepoint-item ::= 'choicepoint' choicept-name':' '{' expression-S '}'.

attribution

choicepoint-item.definitions ← N_definition(gennum,
choicept-name.symbol,
object_definition,
N_mode(choicepoint_item_type,
expression-S.definitions));

A choicepoint splits a transition in two directions. It is uniquely identified by a *choicept-name*. When a choicepoint is encountered, a boolean *expression-S* is evaluated. If it returns true, the transition connected to the *true choicept-end* is taken. If it returns false, the transition connected to the *false choicept-end* is taken. *Choicept-end* will be discussed further on.

rule choicept-name ::= ROOM-identifier.

rule transitions-spec ::= 'transitions' ':' '{' transition-list '}'.

A transition specification is defined by a transitions list.

rule transition-list ::= transition-item ';'.

rule transition-list ::= transition-list transition-item ';'.

attribution

transition-list[1].definitions ← transition-list[2].definitions &
transition-item.definitions.

A *transition-list* is defined by one or more transitions.

rule transition-item ::= 'transition' transition-name ':'
'{' trans-source-pt trans-dest-pt trans-triggers-S trans-code-S '}'.

attribution

transition-item.definitions ← N_definition(gennum,
transition-name.symbol,
object_definition,

N_mode(transition_item_type,
trans-source-pt.definitions,
trans-dest-pt.definitions,
trans-triggers-S.definitions,
trans-code-S.definitions));

condition

unambiguous(trans-triggers-S.definitions);

A transition is the action of going from one state to another. A *transition-name* identifies it uniquely. It has a source point which is the state of origin and a destination point which is the target state. A transition is fired upon certain triggers and some source code can be associated with its triggerring.

rule transition-name ::= ROOM-identifier source-name-S.

attribution

transition-name.symbol ← ROOM-identifier.symbol &
source-name-S.symbol;

A transition name is formed by a *ROOM-identifier* and possibly a source name which identifies where the transition is taken from.

rule source-name-S ::= .

attribution

source-name-S.symbol ← **Nil**;

rule source-name-S ::= '/' source-name.

A *source-name-S* indicates the possibility of source name.

rule source-name ::= state-name-use.

rule source-name ::= choicept-end-use.

rule source-name ::= 'top'.

A source name can be a state name, a *choicept-end* when the transition follows a choicepoint, or *top* when it is the initial transition in its scope.

rule state-name-use ::= state-name.

attribution

state-name-use.definitions ← N_definition(gennum,
state-name.symbol,
unknown_definition);

rule choicept-end-use ::= choicept-end.

attribution

choicept-end-use.definitions ← N_definition(gennum,
choicept-end.symbol,
unknown_definition);

rule trans-source-pt ::= 'source' ':' tr-point-spec.

A transition source point is defined as a transition point specification.

rule trans-dest-pt ::= 'destination' ':' tr-point-spec.

The destination of a transition is defined as a transition point specification.

rule tr-point-spec ::= 'state' state-name-use.

rule tr-point-spec ::= choicept-end-use.

rule tr-point-spec ::= 'initial' 'point'.

rule tr-point-spec ::= 'state' 'border' cont-trans-S.

A transition point specification can be a state name or a *choicept-end.* When it holds the value *initial point*, the transition starts from the initial state.

rule choicept-end ::= 'choicepoint' choicept-name true-false-branch-S.

attribution

choicept-end.definitions ← N_definition(gennum,
choicept-name.symbol & true-false-branch-S.value,

object_definition,

N_mode(choicept_end_type,

true-false-branch.value));

A *choicept-end* identifies the branch that can be taken after a choicepoint evaluation. It is uniquely identified by the combination of a choicepoint name and the value of *true-false-branch.*

rule true-false-branch ::= 'true' 'branch'.

attribution

true-false-branch.value ← true;

rule true-false-branch ::= 'false' 'branch'.

attribution

true-false-branch.value ← false;

A *true-false-branch* indicates the path to take based on the value of the choicepoint expression. It can hold the values *true* or *false.*

rule cont-trans-S ::= .

rule cont-trans-S ::= cont-trans.

A *cont-trans-S* indicates the possibility of a *cont-trans.*

rule cont-trans ::= 'to' 'transition' transition-name-use.

rule cont-trans ::= from-trans-list.

A *cont-trans* is defined as either a transition name or a *from-trans-list.*

rule from-trans-list ::= .

rule from-trans-list ::= from-trans-list 'from' 'transition' transition-name-use.

attribution

from-trans-list[1].definitions ← from-trans-list[2].definitions &

transition-name-use.definitions;

A from-trans-list is defined as list of transition names. This list can be empty.

rule transition-name-use ::= transition-name.

attribution

transition-name-use.definitions ← N_definition(gennum,

transition-name.symbol,

unknown_definition);

rule trans-triggers-S ::= .

rule trans-triggers-S ::= trans-triggers.

A *trans-trigger-S* indicates the possibility of transition triggers.

rule trans-code-S ::= .

rule trans-code-S ::= trans-code.

A *trans-code-S* indicates the possibility of transition source code.

rule trans-triggers ::= 'triggered' 'by' ':' '{' events-list-S '}'.

The triggers that can initiate a transaction are defined in an event list.

rule events-list-S ::= .

rule events-list-S ::= events-list.

An *events-list-S* indicates the prossibility of an event list.

rule events-list ::= event-item.

rule events-list ::= events-list 'or' event-item.

attribution

events-list[1].definitions ← events-list[2].definitions & event-item.symbols;

A event list is composed of one or several events.

rule event-item ::= 'event' ':' '{' 'signals' ':' '{' signals-list '}' 'on' ':' '{' port-saps-list '}'

guard-expression-S '}'.

attribution

event-item.definitions ← signals-list.definitions &

port-saps-list.definitions & guard-expression-S.definitions ;

An event is defined by a list of signals that can trigger it. These signals can come from either ports or service access points. These are listed in *port-saps-list.* When an event occurs, a guard expression can be evaluated to verify that a condition is satisfied before taking the transition.

rule guard-expression-S ::= .

rule guard-expression-S ::= 'guard' ':' '{' bool-expression '}'.

A *guard-expression-S* indicates the possibility of a guard expression.

rule bool-expression ::= expression.

A *bool-epxression* is a function which returns either *true* or *false.*

rule signals-list ::= signal-literal.

rule signals-list ::= signals-list 'or' signal-literal.

attribution

signals-list[1].symbols ← signals-list[2].symbols &
signal-literal.symbol;

A signal list is composed of one or several signals.

rule port-saps-list ::= port-or-sap-name.

rule port-saps-list ::= port-saps-list 'or' port-or-sap-name.

attribution

port-saps-list[1].definitions ← port-saps-list[2].definitions &
port-or-sap-name.definitions;

A *port-saps-list* is composed of one more *port-or-sap-name.*

rule port-or-sap-name ::= port-ref-name.

attribution

port-or-sap-name.definitions ← N_definition(gennum,
port-ref-name.symbol,
unknown_definition);

rule port-or-sap-name ::= sap-ref-name.

attribution

port-or-sap-name.definitions ← N_definition(gennum,
sap-ref-name.symbol,
unknown_definition);

rule trans-code ::= 'code' ':' '{' expression-S '}'.

A *trans-code* is defined as the possibility of an expression. This expression is evaluated when the transition in which it is defined takes place.

A.1.2 Protocol Classes

rule protocol-class-spec ::= 'protocol' 'class' protocol-class-name
'derived' 'from' protocol-superclass-id protocol-spec ';'.

attribution

protocol-class-spec.definitions ← N_definition(gennum,
protocol-class-name.symbol,
type_definition,
N_mode(protocol_class_type,
protocol-superclass-id.mode,
protocol-spec.definitions));

condition

protocol-superclass-id.symbol ≠ protocol-class-name.symbol;

A protocol class specification represents a set messages that are to exchanged between to actors. It is identified by a *protocol-class-name* and can inherit properties from a protocol superclass. The protocol specification gives the list of messages and the direction in which they should travel.

rule protocol-class-name ::= class-name.

rule protocol-superclass-id ::= root-id.

attribution

protocol-superclass-id.mode ← **Nil**;

rule protocol-superclass-id ::= protocol-class-name-use.

A *protocol-superclass-id* can be defined by a protocol-class-name or by *root-id.*

rule protocol-spec ::= '{' service-spec messages-spec '}'.

attribution

protocol-spec.definitions ← service-spec.value & messages-spec.definitions;

A *protocol-spec* is defined by a service specification and message specifications.

rule service-spec ::= 'service' ':' service-name.

A *service-spec* is defined by a service name.

rule messages-spec ::= 'message' 'types' ':' '{' in-messages-spec out-messages-spec '}'.

attribution

messages-spec.definition ← in-messages-spec.definitions &

out-messages-spec.definitions;

condition

unambiguous(in-messages-spec.definitions) **and**

unambiguous(out-messages-spec.definitions);

A *messages-spec* is defined by incoming and outgoing messages specifications.

rule in-messages-spec ::= 'in' ':' '{' message-types-list '}'.

An *in-messages-spec* is defined as a message type list.

rule out-messages-spec ::= 'out' ':' '{' message-types-list '}'.

An *out-messages-spec* is defined as a message type list.

rule message-types-list ::= message-type-spec ';'.

rule message-types-list ::= message-types-list message-type-spec ';'.

attribution

message-types-list[1].definitions ← message-types-list[2].definitions &

message-type-spec.definitions ;

A *messages-types-list* is defined by one or several message type specification.

rule message-type-spec ::= '{' 'signal' ':' signal-literal

'data' 'class' ':' data-class-name-use '}'.

attribution

message-type-spec.definitions← N_definition(gennum,

signal-literal.symbol,

type_definition,

data-class-name-use.mode)) ;

A message type specification defines a message. It is uniquely identified by a signal literal and a data object of the type symbolized by *data-class-name-use*.

rule signal-literal ::= '%'signal-name.

A *signal-literal* is defined as a signal name.

rule signal-name ::= ROOM-identifier.

A.1.3 Data Classes

rule data-class-spec ::= data-spec ';'.

rule data-class-spec ::= constant-spec ';'.

A data class specification can be either a data specification or a constant specification.

rule data-spec ::= 'data' 'class' data-class-name 'derived' 'from' data-superclass-id

'isa' data-type-spec methods-spec-S.

attribution

data-spec.definitions ← N_definition(gennum,

data-class-name.symbol,

type_definition,

N_mode(data_spec_type,

data-class-superclass-id.definitions,

data-type-spec.mode,

methods-spec-S.definitions));

condition

unambiguous(methods-spec-S.definitions);

A data specification defines a data class. It is uniquely identified by a *data-class-name* and can inherit from a data superclass. Its data structure is defined by a data type specification and *methods-spec-S* gives the possibility to specify methods to manipulate the data in the class.

rule data-class-name ::= class-name.

A data class name is defined as a *class-name.*

rule data-superclass-id ::= root-id.

attribution

data-superclass-id.definitions ← **Nil**;

rule data-superclass-id ::= data-class-name-use.

A *data-superclass-id* give the parent of a data class. When it holds a data class name, the target class inherits all the properties of that class. When it holds the *root-id*, it inherits from no other classes.

rule data-type-spec ::= choice-type-spec.

rule data-type-spec ::= enumer-spec.

rule data-type-spec ::= sequence-spec.

rule data-type-spec ::= seq-of-spec.

rule data-type-spec ::= integer-spec.

rule data-type-spec ::= real-spec.

rule data-type-spec ::= string-spec.

rule data-type-spec ::= undef-type-name.

rule choice-type-spec ::= 'Choice' '{' type-names-list-S '}'.

attribution

choice-type-spec.mode ← N_mode(choice_type_spec_type,
type-names-list-S.modes);

rule type-names-list-S ::= .

rule type-names-list-S ::= type-names-list.

A *type-names-list-S* indicates the possibility of a type name list.

rule type-names-list ::= data-class-name-use ';'.

rule type-names-list ::= type-names-list data-class-name-use ';'.

attribution

type-names-list[1].modes ← type-names-list[1].modes &
data-class-name-use.modes ;

rule enumer-spec ::= 'Enumerated' '{' enumer-list-S '}'.

attribution

enumer-spec.mode ← N_mode(enumer_spec_type, enumer-list-S.symbols);

rule enumer-list-S ::= .

rule enumer-list-S ::= enumer-list.

An *enumer-list-S* indicates the possibility of an enumeration list.

rule enumer-list ::= enumer-literal ';'.

rule enumer-list ::= enumer-list enumer-literal ';' .

attribution

enumer-list[1].symbols ← enumer-list[2].symbols &

```
        enumer-literal.symbol;

rule enumer-literal ::= ROOM-identifier.

rule sequence-spec ::= 'Sequence' '{' seq-field-list-S '}'.
attribution
    sequence-spec.mode ← N_mode(sequence_spec_type, seq-field-list-S.definitions

rule seq-field-list-S ::= .
rule seq-field-list-S ::= seq-field-list.

rule seq-field-list ::= seq-field-spec ';'.
rule seq-field-list ::= seq-field-list seq-field-spec ';'.
attribution
    seq-field-list[1].definitions ← seq-field-list[1].definitions &
        seq-field-spec.definitions

rule seq-field-spec ::= field-name 'isa' data-class-name-use optional-S default-S.
attribution
    seq-field-spec.definitions ← N_definition(gennum,
        field-name.symbol,
        object_definition,
        N_mode(seq_field_spec_type,
                data-class-name-use.mode,
                optional-S.value,
                default-S.value));

rule field-name ::= ROOM-identifier.

rule optional-S ::= .
attribution
    optional-S.value ← Nil;
```

rule optional-S ::= 'optional'.

attribution

optional-S.value ← optional;

An *optional-S* indicates whether the field is optional or not.

rule default-S ::= .

attribution

default-S.value ← **Nil**;

rule default-S ::= 'default' '{' value-literal '}'.

attribution

default-S.value ← value-literal.symbol;

A *default-S* indicates the possibility of a default value.

rule seq-of-spec ::= 'SequenceOf' replication-factor data-class-name-use.

attribution

seq-of-spec.mode ← N_mode(seq_of_spec_type,
replication-factor.value,
data-class-name-use.mode);

rule integer-spec ::= 'Integer' int-minimum-S int-maximum-S.

attribution

integer-spec.mode ← N_mode(integer_spec_type,
int-minimum-S.value,
int-maximum-S.value);

condition

(int-minimum-S.value < int-maximum-S.value);

rule int-minimum-S ::= .

attribution

int-minimum-S.value ← **Nil**;

rule int-minimum-S ::= minimum '{' integer '}'.

attribution

int-minimum-S.value ← integer.symbol;

A *int-minimum-S* indicates the possibility of a minimum limit.

rule int-maximum-S ::= .

attribution

int-maximum-S.value ← **Nil**;

rule int-maximum-S ::= 'maximum '{' integer '}'.

attribution

int-maximum-S.value ← integer.symbol;

A *int-maximum-S* indicates the possibility of a maximum limit.

rule real-spec ::= 'Real' minimum-S maximum-S.

attribution

real-spec.mode ← N_mode(real_spec_type,
real-minimum-S.value,
real-maximum-S.value);

condition

real-minimum-S.value < real-maximum-S.value;

rule real-minimum-S ::= .

attribution

real-minimum-S.value ← **Nil**;

rule real-minimum-S ::= minimum '{' real '}'.

attribution

real-minimum-S.value ← real.symbol;

A *real-minimum-S* indicates the possibility of a minimum limit.

rule real-maximum-S ::= .

attribution

real-maximum-S.value ← **Nil**;

rule real-maximum-S ::= 'maximum '{' real '}'.

attribution

real-maximum-S.value ← real.symbol;

A *real-maximum-S* indicates the possibility of a maximum limit.

rule string-spec ::= 'String' replication-factor 'numeric'.

attribution

string-spec.mode ← N_mode(string_numeric_spec_type,
replication-factor.value);

rule string-spec ::= 'String' replication-factor 'printable'.

attribution

string-spec.mode ← N_mode(string_printable_spec_type,
replication-factor.value);

rule undef-type-name ::= 'Unspecified'.

attribution

undef-type-name.mode ← N_mode(undef_type_name_type);

rule constant-spec ::= 'constant' constant-name isa-const-type-spec-S.

attribution

constant-spec.definitions ← N_definition(gennum,
constant-name.symbol,
object_definition,
N_mode(constant_spec_type,
isa-const-type-spec-S.mode);

rule constant-name ::= ROOM-identifier.

rule isa-const-type-spec-S ::= .

rule isa-const-type-spec-S ::= 'isa' const-type-spec.

An *isa-const-type-spec-S* indicates the possibility of a constant type specification.

rule const-type-spec ::= valid-const-type 'value' '{' constant-literal '}'.

attribution

const-type-spec.mode ← N_mode(const_type_spec_type,
valid-const-type.mode,
constant-literal.value);

rule methods-spec-S ::= .

rule methods-spec-S ::= methods-spec.

A *methods-spec-S* indicates the possibility of a methods specification.

rule methods-spec ::= 'methods' 'functions' ':' instance-methods-spec-S
class-methods-spec-S.

attribution

methods-spec.definitions ← instance-methods-spec-S.definitions &
class-methods-spec-S.definitions;

condition

unambiguous(instance-methods-spec-S.definitions &
class-methods-spec-S.definitions);

A methods specification give the possibility to specify instance methods and class methods.

rule instance-methods-spec-S ::= .

rule instance-methods-spec-S ::= instance-methods-spec.

An *instance-methods-spec-S* indicates the possibility of an instance methods specification.

rule class-methods-spec-S ::= .

rule class-methods-spec-S ::= class-methods-spec.

A *class-methods-spec-S* indicates the possibility of a class methods specification.

rule instance-methods-spec ::= 'instance' ':' '{' functions-list '}'.

An *instance-methods-spec* is defined as a list of function which are called upon instances of the class.

rule class-methods-spec ::= 'class' ':' '{' functions-list '}'.

A *class-methods-spec* is defined as a list of functions which are called upon the class itself.

Bibliographie

[1] J. W. De Bakker. *Formal Definitions of Programming Languages With an Application to the Definition of ALGOL 60.* Math Cent. Tracts 16, Mathematisch Centrum, Amsterdam, 1967.

[2] G. Booch. *Object-Oriented Analysis and Design with Applications.* Benjamin / Cummings Publishing Company, Inc. Second Edition, 1994.

[3] P. Coad, E. Yourdon. *Object-Oriented Analysis.* Prentice-Hall, Second Edition, 1991.

[4] P. Coad, E. Yourdon. *Object-Oriented Design.* Prentice-Hall, 1991.

[5] M. Davidoff. *The Satellite Experimenter's Handbook.* The American Radio Relay League. Second Edition, 1994.

[6] T. DeMarco. *Structured Analysis and System Specification.* Yourdon Press, 1978.

[7] A. Goldberg, D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[8] J. Grosch. *Rex - A Scanner Generator.* Compiler Generation Report No. 5, GMD Forschungsstelle an der Universität Karlsruhe, July, 1992.

[9] D. Harel. *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming 8, July, 1987.

[10] D. Harel, A. Pnueli, J.P. Schmidt, R. Sherman. *On the formal semantics of Statecharts.* Proceedings of the Second IEEE Symposium on Logic in Computer Science, IEEE Press, 1987.

[11] I. Jacobson. *Object-Oriented Software Engineering: A Use Case Driven Approach.* ACM Press Addison Wesley, 1992.

[12] D. E. Knuth. *Semantics of Context-Free Languages.* Mathematical Theory, Vol. 2, No. 2, 1968.

[13] P. J. Landin. *The Mathematical Evaluation of Expressions.* Comp. J. 6, 1964.

[14] P. J. Landin. *A Correspondance Between ALGOL 60 and Church's Lambda Notation.* Comm. ACM 6, 1965.

[15] J. R. Levine, T. Mason, D. Brown. *lex & yacc.* O'Reilly & Associates, Inc. Second Edition, 1995.

[16] ITU-T. *Draft Recommendation Z.120, Message Sequence Chart (MSC)* COM X-R 22-E, Geneva, March 1992.

[17] M. Normandeau and M. Barbeau. *Object-Oriented Modeling of a Satellite Tracking Software.* 15th ARRL and TAPR Digital Communications Conference, Seattle, 1996.

[18] *The ObjecTime Toolset Reference Guide* ObjecTime Ltd, 1994.

[19] *The ObjectGEODE Toolset Reference Guide* Verilog SA, 1996.

[20] P. Leblanc, V. Encontre. *Object-Oriented Real-Time Techniques: Method Guidelines* Version 1.0, Verilog, 1996.

[21] QNX Software Systems Ltd. *QNX System Architecture.* 1993.

[22] J. Röhrich. *Syntax-Error Recovery in LR-Parsers.* Informatik-Fachberitchte, vol.1, H.-J. Schneider and M. Nagl (ed.), Springer-Verlag, Berlin, 1976.

[23] J. Röhrich. *Methods for the Automatic Construction of Error Correcting Parsers.* Acta Inf. 13,2, 1980.

[24] W.W. Royce. *Managing the Development of Large Software Systems: Concepts and Techniques.* Proceedings of WESCON, August 1970.

[25] J. Rumbaugh, et al. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[26] B. Selic, G. Gullekson, and P. T. Ward. *Real-time Object-Oriented Modeling.* John Wiley and Sons, Inc. 1994.

[27] B. Stroustrup. *The C++ Programming Language 2nd edition.* Addison-Wesley, 1991.

[28] G. Booch, I. Jacobson, J. Rumbaugh. *The Unified Modeling Language for Object-Oriented Development.* Documentation Set Version 1.0, January 1997.

[29] G. Booch, I. Jacobson, J. Rumbaugh. *Unified Modeling Language For Real-Time Systems Design.* Version 2.0, 1996.

[30] ITU-T. *Recommendation Z.100, Specification and Description Language (SDL)* COM X-R 17-E, Geneva, March 1992.

[31] W. M. Waite, G. Goos. *Compiler Construction.* Springer-Verlag, 1985.

[32] E. Yourdon. *Modern Structured Analysis.* Yourdon Press, 1989.